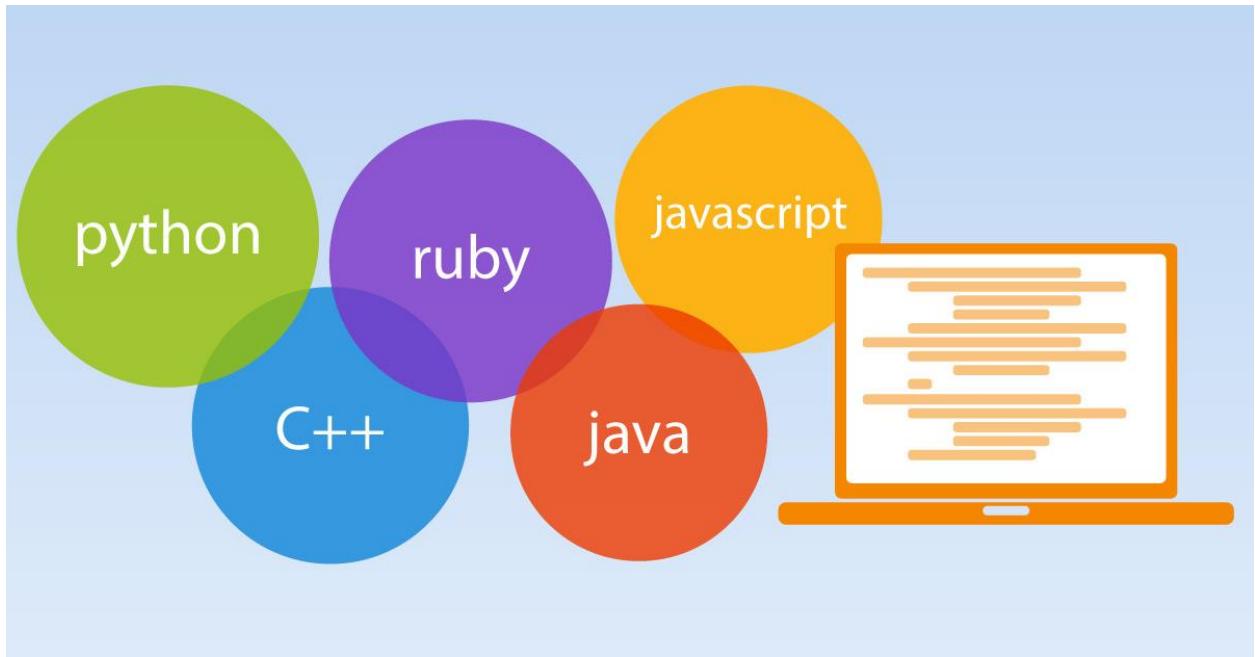


Research Papers on Programming Languages



These papers provide a breadth of information about Programming Languages that is generally useful and interesting from a computer science perspective.



Contents

- Linear Logic
- Gradual Typing for Functional Languages
- Soft Typing
- Linear Types Can Change The World!
- Separation Logic and Abstraction
- Separation Logic, Abstraction and Inheritance
- An Object-Oriented Effects System
- A Type System for Borrowing Permissions
- A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language
- Promises: Limited Specifications for Analysis and Manipulation
- Gradual Typing for Objects

- Capabilities for Uniqueness and Borrowing
- A Verified Compiler for an Impure Functional Language
- Data groups: Specifying the modification of extended state
- Separation and Information Hiding
- Separation Logic: A Logic for SharedMutable Data Structures
- Modular Typestate Checking of Aliased Objects

LINEAR LOGIC*

Jean-Yves GIRARD

Équipe de Logique Mathématique, UA 753 du CNRS, UER de Mathématiques, Université de Paris VII, 75251 Paris, France

Communicated by M. Nivat

Received October 1986

A la mémoire de Jean van Heijenoort

Abstract. The familiar connective of negation is broken into two operations: linear negation which is the purely negative part of negation and the modality "of course" which has the meaning of a reaffirmation. Following this basic discovery, a completely new approach to the whole area between constructive logics and programmation is initiated.

Contents

I. Introduction and abstract	2
II. Linear logic explained to a proof-theorist	4
II.1. The maintenance of space in sequent calculus	4
II.2. Linear logic as a sequent calculus	4
II.3. Strength of linear logic	5
II.4. Subtlety of linear logic	6
II.5. The semantics of linear logic: phases	6
III. Linear logic explained to a (theoretical) computer scientist	7
III.1. The semantics of linear logic: coherent spaces	8
III.2. Proof-nets: a classical natural deduction	8
III.3. Normalization for proof-nets	11
III.4. Relevance for computer science	12
III.4.1. Questions and answers	12
III.4.2. Towards parallelism	12
III.4.3. Communication and trips	13
III.4.4. Work in progress	14
IV. Pons asinorum: from usual implication to linear implication	14
IV.1. Interpretation of functional languages	15
IV.2. The disturbance	15
IV.3. The decomposition	16
IV.4. Further questions	17
1. The phase semantics	17

* Because of its length and novelty this paper has not been subjected to the normal process of refereeing. The editor is prepared to share with the author any criticism that eventually will be expressed concerning this work.

2. Proof-nets	28
2.1. The concept of trip	30
2.2. The cut rule	40
2.3. Cut elimination	41
2.4. The system PN1	44
2.5. The system PN2	46
2.6. Discussion	47
3. Coherent semantics	48
4. Normalization in PN2	60
4.1. Axiom contraction	61
4.2. Symmetric contractions	62
4.3. Commutative contractions	65
5. Some useful translations	78
5.1. The translation of intuitionistic logic	78
5.2. The translation of the system F	82
5.2.1. Types	82
5.2.2. Translation of terms	82
5.2.3. Semantic soundness of the translation	84
5.2.4. Syntactic soundness of the translation	85
5.3. The translation of current data in PN2	85
5.3.1. Booleans	86
5.3.2. Integers	86
5.3.3. Lists, trees	90
5.4. Interpretation of classical logic	90
5.5. Translation of cut-free classical logic	91
5.6. The Approximation Theorem	92
6. Work in progress: slices	93
V. Two years of linear logic: selection from the garbage collector	97
V.1. The first glimpses	97
V.2. The quantitative attempt	98
V.3. The intuitionistic attempt	98
V.4. Recent developments	99
V.5. The exponentials	100
Acknowledgment	101
References	102

“La secrète noirceur du lait...”

(Jacques Audiberti)

i. Introduction and abstract

Linear logic is a logic behind logic, more precisely, it provides a continuation of the constructivization that began with intuitionistic logic. The logic is as strong as the usual ones, i.e., intuitionistic logic can be translated into linear logic in a faithful way. Linear logic shows that the constructive features of intuitionistic operations are indeed due to the linear aspects of some intuitionistic connectives or quantifiers, and these linear features are put at the prominent place. Linear logic has two

semantics:

(i) a Tarskian semantics, the *phase semantics*, based on the idea that truth makes an angle with reality; this ‘angle’ is called a phase, and the connectives are developed according to those operations that are of immediate interest in terms of phases.

(ii) a Heytingian semantics, the *coherent semantics*, which is just a cleaning of familiar Scott semantics for intuitionistic operations, followed by the decomposition of the obtained operations into more primitive ones.

The connectives obtained are

- the *multiplicatives* (\otimes, \wp, \multimap) which are bilinear versions of “and”, “or”, “implies”;
- the *additives* ($\oplus, \&$) which are linear versions of “or”, “and”;
- the *exponentials* ($!, ?$) which have some similarity with the modals \Box and \Diamond and which are essential to preserve the logical strength.

All these connectives are dominated by the linear negation $(.)^\perp$, which is a constructive and involutive negation; by the way, linear logic works in a classical framework, while being more constructive than intuitionistic logic. The phase semantics is complete w.r.t. linear *sequent calculus* which is, roughly speaking, sequent calculus without weakening and contraction. Since the framework is classical, it is necessary, in order to get a decent proof-theoretic structure, to consider *proof-nets* which are the natural deduction of linear logic; i.e., a system of proofs with multiple conclusions, which works quite well.

One of the main outputs of linear logic seems to be in computer science:

(i) As long as the exponentials $!$ and $?$ are not concerned, we get a very sharp control on normalization; linear logic will therefore help us to improve the efficiency of programs.

(ii) The new connectives of linear logic have obvious meanings in terms of parallel computation, especially the multiplicatives. In particular, the multiplicative fragment can be seen as a system of communication without problems of synchronization. The synchronization is handled by proof-boxes which are typical of the additive level. Linear logic is the first attempt to solve the problem of parallelism *at the logical level*, i.e., by making the success of the communication process only dependent of the fact that the programs can be viewed as *proofs* of something, and are therefore sound.

(iii) There are other potential fields of interest, e.g., databases: the use of classical logic for modelling automatic reasoning has led to logical atrocities without any practical output. One clearly needs more subtle logical tools taking into account that it costs something to make a deduction, a guess, etc.; linear logic is built on such a principle (the phases) and could serve as a prototype for a more serious approach to this subject.

(iv) The change of logic could change the possibilities of logical programming since linear negation allows a symmetric clausal framework, free from the usual problems of classical sequent calculus.

II. Linear logic explained to a proof-theorist

There is no constructive logic beyond intuitionistic logic: *modal logic*, inside which intuitionistic connectives can be faithfully translated, is nonconstructive; various logics which have been considered in the philosophical tradition lack the seriousness that is implicitly needed (a clean syntax, cut-elimination, semantics, etc.) and, in the best case, they present themselves as impoverishments of intuitionism, while we are of course seeking something as strong as intuitionism but more subtle. The philosophical exegesis of Heyting's rules leaves in fact very little room for a further discussion of the intuitionistic calculus; but has anybody ever seriously tried? In fact, linear logic, which is a clear and clean extension of usual logic, can be reached through a more perspicuous analysis of the semantics of proofs (not very far from the computer-science approach and thus relegated to the next section), or by certain more or less immediate considerations about sequent calculus. These considerations are of immediate geometrical meaning but in order to understand them, one has to forget the intentions, remembering, with a Chinese leader, that it is not the colour of the cat that matters, but the fact it catches mice.

II.1. The maintenance of space in sequent calculus

When we write a sequent in classical logic, in intuitionistic or minimal logic, the only difference is the *maintenance of space*: in classical logic we have $n+m$ rooms separated by \vdash ; in minimal logic $m=1$, while in intuitionistic logic $m=0$ or 1. Into the three cases we build particular connectives which belong to such and such tradition, but beyond any tradition we are extremely free, provided we respect an implicit symmetry that is essential for cut-elimination.

Now, what is the meaning of the separation \vdash ? The classical answer is "to separate positive and negative occurrences". This is factually true but shallow; we shall get a better answer by asking a better question: what in the essence of \vdash makes the two latter logics more constructive than the classical one? For this the answer is simple: take a proof of the existence or the disjunction property; we use the fact that the last rule used is an introduction, which we cannot do classically because of a possible contraction. Therefore, in the minimal and intuitionistic cases, \vdash serves to mark a place where contraction (and maybe weakening too) is forbidden; classically speaking, the \vdash does not have such a meaning, and this is why lazy people very often only keep the right-hand side of classical sequents. Once we have recognized that the constructive features of intuitionistic logic come from the dumping of structural rules on a specific place in the sequents, we are ready to face the consequences of this remark: the limitation should be generalized to the other rooms, i.e., weakening and contraction disappear. As soon as weakening and contraction have been forbidden, we are in linear logic.

II.2. Linear logic as a sequent calculus

Linear logic ignores the left/right-asymmetry of intuitionism; in particular, one can directly deal with right-handed sequents. The first thing to do is to try to rewrite

familiar connectives in this framework where structural rules have been limited to exchange; in particular, the two possible traditions for writing the right \wedge -rule:

$$\frac{\vdash A, C \quad \vdash B, D}{\vdash A \wedge B, C, D} M, \quad \frac{\vdash A, C \quad \vdash B, C}{\vdash A \wedge B, C} A,$$

which are equivalent in classical sequent calculus modulo easy structural manipulations, now become two radically different conjunctions:

- rule (M) treats the contexts by juxtaposition and yields the *multiplicative conjunction* \otimes (*times*);
- rule (A) treats the contexts by identification, and yields the *additive conjunction* $\&$ (*with*).

The vaultkey of the system is surely *linear negation* $(.)^\perp$. This negation, although constructive, is involutive! All the desirable De Morgan formulas can be written with $(.)^\perp$, without losing the usual constructive features. Through the De Morgan translations, the other elementary connectives are easily accessible: the multiplicative disjunction \wp (*par*) and the additive one \oplus (*plus*) and linear implication \multimap (*entails*). The last connective was the first to find its way into official life and it has given its name to the full enterprise.

There is a philosophical tradition of ‘strict implication’ amounting to Lewis. In some sense, linear implication agrees with this tradition: in a linear implication, the premise is used ‘once’, in the sense that weakening and contraction are forbidden on the premise (in fact ‘once’ means only something in multiplicative terms, i.e., w.r.t. juxtaposition: additively speaking, the premise can be used several times. $A \multimap A \otimes A$ is not derivable, but $A \multimap A \& A$ is). Even if this philosophical tradition has not been very successful, the existence of linear logic gives a retrospective justification to these attempts.

The most hidden of all linear connectives is *par*, which came to light purely formally as the De Morgan dual of \otimes and which can be seen as the effective part of a classical disjunction. Typically, $A \multimap A$, which everybody understands, is literally the same as $A^\perp \wp A$.

II.3. Strength of linear logic

Is linear logic strong enough? In other terms, is it possible to translate usual logic (especially intuitionistic logic) into linear logic? If we look at usual connectives, we discover that some laws belong to the multiplicative universe (e.g., $A \multimap A$), and others to the additive realm (e.g., the equivalence between A and $A \& A$). In fact, no translation works, for a very simple reason: since there are no structural rules, the proofs diminish in size during normalization! This feature, one of the most outstanding qualities of linear logic, definitely forbids any decent translation on the basis of the connectives so far written. The only solution is to allow weakening and contraction to some extent; in order to do so, the *exponentials* ! and ? are introduced; these modalities indicate the possibility of structural rules on the formula beginning with them. For linear proofs involving these modalities, a loss of control over the

normalization times occurs, as expected. Roughly speaking, one can say that usual logic (which can be viewed as the adjunction of the modalities) appears as a passage to the limit in linear logic since, for $?A$ to be equivalent with $?A \wp ?A$, we have to do something like an infinite “par”. The translation of intuitionistic logic into linear logic is defined by

$$\begin{aligned} A^0 &= A \quad \text{for } A \text{ atomic,} \\ (A \wedge B)^0 &= A^0 \& B^0, \quad (A \vee B)^0 = !A^0 \oplus !B^0, \quad (A \Rightarrow B)^0 = !A^0 \multimap B^0, \\ (\neg A)^0 &= !A^0 \multimap \mathbf{0} \quad (\mathbf{0} \text{ is one of the constants of the system}), \\ (\forall x.A)^0 &= \bigwedge x.A^0, \quad (\exists x.A)^0 = \bigvee x.!A^0 \end{aligned}$$

(\wedge and \vee are the linear quantifiers, about which there is little to say except that \vee is effective, and the De Morgan dual of $\wedge !$).

II.4. Subtlety of linear logic

Let us give an example in order to show the kind of unexpected distinctions that linear logic may express; for this, we shall assume that the formula A is quantifier-free, and is represented in linear logic in such a way that “or” between instances of A becomes “ \wp ”. We claim that linear logic can write a formula with this meaning: $\exists x A \wp y$ and this with a Herbrand expansion of length 2. Intuitionistic logic can handle the Herbrand expansion of length 1 and classical logic can handle the general case with no bound on the length, but the much more interesting case of a given length bound was not expressible in usual logics. People with some proof-theoretic background know that a Herbrand expansion of length 2 can be rewritten as a sequent $\vdash A t x, A u x'$, with x, x' not free in t , x' not free in u (midsequent theorem). Now, the formula

$$(\bigvee x \wedge y A \wp y) \wp (\bigvee x \wedge y A \wp y)$$

of linear logic will be cut-free provable exactly when a sequent of the above kind is provable.

II.5. The semantics of linear logic: phases

The sentence saying that usual logic is obtained from linear logic by a passage to the limit is reminiscent of the relation between classical and quantum mechanics. There is a Tarskian semantics for linear logic with some physical flavour: w.r.t. a certain monoid of *phases* formulas are true in certain situations. The set of all phases for which a formula may be true is called a *fact*, among which the fact of *orthogonal phases* plays a central role to represent the absurdity \perp . One easily defines the *orthogonal* A^\perp of a fact and by restricting to facts which are themselves orthogonals, we get an involutive operation representing linear negation. There usual connectives are easily defined: $\&$ is the intersection, whereas \multimap is the *dephasing*, i.e., $A \multimap B$ is the set of all p such that $p.A \subset B$, where the phase space, although commutative, is

being written multiplicatively. A fact is said to hold when it contains the unit phase 1. This semantics is complete and sound w.r.t. linear logic. One of the wild hopes that this suggests is the possibility of a direct connection with quantum mechanics . . . but let's not dream too much!

III. Linear logic explained to a (theoretical) computer scientist

There are still people saying that, in order to make computer science, one essentially needs a soldering iron; this opinion is shared by logicians who despise computer science and by engineers who despise theoreticians. However, in recent years, the need for a logical study of programmation has become clearer and clearer and the linkage logic-computer-science seems to be irreversible.

(i) For computer science, logic is the only way to rationalize '*bricolage*'; for instance, what makes resolution work is not this or that trick, but the fundamental results on sequent calculus. More: if one wants to improve resolution, one has to respect the hidden symmetries of logic. In similar terms, the use of typed systems (e.g., the system *F*), is a safeguard against errors of various sorts, e.g., loops or bugs. Moreover, in domains where the methodology is a bit hesitating, as in parallelism, logic is here as the only milestone. In some sense, logic plays the same role as the one played by geometry w.r.t. physics: the geometrical frame imposes certain conservation results, for instance, the Stokes formula. The symmetries of logic presumably express deep conservation of information, in forms which have not yet been rightly conceptualized.

(ii) For logic, computer science is the first real field of application since the applications to general mathematics have been too isolated. The applications have a feedback to the domain of pure logic by stressing neglected points, shedding new light on subjects that one could think of as frozen into desperate staticism, as classical sequent calculus or Heyting's semantics of proofs. Linear logic is an illustration of this point: everything has been available to produce it since a very long time; in particular, retrospectively, the syntactic restriction on structural rules seems so obviously of interest that one can hardly understand the delay of fifty years in its study. Computer science prompted this subject through semantics: The idea behind Heyting's semantics of proofs is that proofs are functions; this idea has been put into a formal correspondence between proofs and functional systems by Curry, Howard, and De Bruijn. The tradition on Heyting's semantics is of little interest because it is full of theological distinctions of the style: "how do we prove that this a proof", etc.; computer scientists have attacked more or less the same question, but with the idea of concentrating on the material part of these proofs, or functions. This has led to Scott semantics, for long the reference concept in the subject; however, Scott's semantics is not free from big defects, specifically, where what is called *finite* in these domains is not finite in any reasonable sense (*noetherian* would have been a happier terminology) and in particular, the fact that arbitrary

objects are approximable by finite ones only because we have abused the word ‘finite’. In recent work in 1985, the present author has made essential simplifications of Scott’s work: by use of a more civilized approach to function spaces one can work with qualitative semantics (rebaptized here to *coherent spaces*) in which finiteness is finiteness. Moreover, all basic definitions involving coherent spaces are very simple and a certain number of features that were hidden behind the heavy apparatus of Scott domains are brought to light, in particular, the discovery that *the arrow is not primitive*.

III.1. The semantics of linear logic: coherent spaces

A *coherent space* is a graph on a set (the *web* of the space). What counts as an object in this space is any subset of the graph made of pairwise compatible (we say: *coherent*) points. The interpretations made in 1984–85 gave very simple interpretations for the intuitionistic connectives in terms of coherent spaces as schemes for building new coherent spaces. Now, it turns out that there are intermediate constructions; typically, the function space $X \Rightarrow Y$ can be split into

- the formation of the ‘repetition space’ $!X = X'$;
- the formation of the linear implication $X' \multimap Y$.

Later, \multimap can in turn be split as $X \multimap Y = X^\perp \wp Y$, but this is a bit more dubious since $X \wp Y = X^\perp \multimap Y$, so there is some subjectivity in deciding which among the two multiplicatives is more primitive (not to speak of \otimes !).

In fact, after some times, one arrives at the following stock of operations: $(.)^\perp, \wp, \multimap, \otimes, \&, \oplus, !, ?$; usual functional types admit decompositions according to the formulas in Section II.3. One can imagine other linear connectives (e.g., a primitive equivalence) which, although very natural, do not lead to any logical system. What is outstanding here is that the decomposition can also be carried out *at the logical level*; i.e., that these new connectives have all qualities of a logic. This logic has already been disclosed before, but we have to look again at its proof-system, as will be done in the following section.

III.2. Proof-nets: a classical natural deduction

In the beginning (say: up to the end of ’85) things remained simple because it was not clear that the intuitionistic framework was an artificial limitation; at that moment, $(.)^\perp$, \wp , and $?$ were still in *statu nascendi*. It was therefore possible to keep a functional notation for the proofs of this intuitionistic linear logic: for instance, the formation of the linear application tu was subject to the restriction $\text{FV}(t) \cap \text{FV}(u) = \emptyset$; for the additive pair $\& tu$, the restriction was $\text{FV}(t) = \text{FV}(u)$, for linear lambda abstraction $Lx.t$, the restriction was $x \in \text{FV}(t)$, etc. This gave a reasonable system, still of interest, however with the defect of too many connectives involving commutative conversions ($\otimes, \oplus, !$). The conviction that the system should be relevant to parallelism and the rough analogy ‘sequential = intuitionistic, parallel = classical’ produced a shift to the ‘classical’ framework, where sequents are left/right-symmetric

with bad consequences for normalization since it is well-known that no decent natural deduction exists in classical frameworks. However, linear logic is not so bad since it has been possible to define a notation of ‘*proof-net*’ which is a proof with several conclusions. (It seems that the problem is hopeless in usual classical logic and the accumulation of several inconclusive attempts is here to back up this impression. In linear logic, we can distinguish between, for instance, the multiplicative features of disjunction, which deserve a certain treatment, and the additive ones, which deserve another treatment; trying to unify both treatments would lead to a mess.)

In fact, the core of proof-nets consists in studying the multiplicative fragment, which is handled in a very satisfactory way: starting with axioms $A \quad A^\perp$ one develops proof structures by combination of two binary rules, or *links*,

$$\frac{A \quad B}{A \otimes B} \quad \text{and} \quad \frac{A \quad B}{A \wp B}.$$

Most of these structures are incorrect because the left-hand side may work in a way contradicting the right-hand side. Now the soundness criterion is as follows: to each multiplicative link is associated a *switch* with two positions “L” and “R”; an ideal particle (representing the flow of input/output inside the proof) tries to travel starting from a given formula, and in a given direction (upwards: question, or downwards: answer); at each moment, the particle knows (the switches are here to help) where to go, so it eventually makes a cyclic *trip*.

The soundness requirement is simple: for any positioning of the n switches, the trip is long, i.e., it goes twice through all formulas, once up, once down; in other terms, the whole structure has been visited, and every question has been answered, every answer has been questioned. Clearly, the most difficult result of the paper is to prove that this notion of proof-net is equivalent to a sequent calculus approach, i.e., can be sequentialized. The full calculus does not have such a system: it is handled through the concept of *proof-box*, which is a lazy device: moments of sequentialization are put into boxes and the boxes are interconnected in the perfect multiplicative way already explained.

The soundness conditions for proof-nets (absence of shorttrips) should not be misunderstood: it is not a condition that should be practically verified because, with n switches, we essentially have 2^n verifications to make. It must be seen as an *abstract* property of these parallel proofs that we are not supposed to check by means of a concrete algorithm. (In the same way, the fact that an object of type $\text{int} \Rightarrow \text{int}$ of F when applied to an integer \bar{n} yields, after normalization, an integer \bar{m} need not be checked; this, however, is an important abstract property of F). However, we want to work with proof-nets as programs, and when we have one, we want to be sure by some means that it is actually a proof-net; this is why methods for generating proof-nets are very important, and we know the following ones:

(i) *Desequentialization* of a proof in linear sequent calculus: we can imagine this as the most standard way to obtain a proof-net. The user would produce a

source program (in linear sequent calculus), and this program would be compiled as a proof-net, i.e., as a graph between occurrences of formulas.

(ii) *Logical operations* between proof-nets: for instance, a proof-net β with two conclusions A and B can be connected to a proof-net β' with conclusion A^\perp , by means of the cut-link in Fig. 1(a). Here, β has been seen as a (linear) function taking as arguments objects of type A^\perp (proof-nets admitting A^\perp as conclusion) and yielding an answer of type B . The function applies to the argument by means of CUT.



Fig. 1.

The picture is however wholly symmetrical; one can also apply β to an argument β'' of type B^\perp , and this yields a result of type A as shown in Fig. 1(b). One can also combine both ideas and apply β to both β' and β'' etc.; when we say that we ‘apply’ β to β' , this is really to explain the things very elementarily because one could have said that β' is applied to β . There is a perfect function/argument symmetry in linear logic.

(iii) *Normalization*: There is a normalization procedure which eliminates the use of the cut-link. When we normalize a proof-net, then the result is still a proof-net, as we can easily prove. In the study of normalization, the structure

$$\frac{A \quad A^\perp}{\text{CUT}}$$

plays an important role: first, this is not a proof-net, because there is the shorttrip $A^\wedge, A^\perp, A^\wedge$ etc.; second, it would be a ‘black hole’ in the normalization process, i.e., a structure hopelessly normalizing into itself. If we try to imagine what could be the meaning of this pattern, think of a variable of type A (denoting the address of a memory case) calling itself. To some extent, the conditions on trips are here just to forbid this ‘black hole’ and to forbid by the way all situations that would lead to such a pattern by normalization, etc.

Although we can deal with concrete proof-nets without a *feasible* characterization, alternative definitions of soundness are welcome. One can expect from them two kinds of improvements:

- (i) feasibility, i.e., the possibility to use proof-nets directly that have been written on a screen for instance;
- (ii) the extension of the advantages of the parallel syntax outside the multiplicative fragment; for instance, the weakening rules for (\perp) and $(?)$ are handled in terms

of boxes because our definition of rip excludes disconnected nets. Surely, some disconnected nets are however acceptable, but we have not found any simple soundness characterization for them.

III.3. Normalization for proof-nets

There is a radical and quick procedure for normalizing the multiplicative fragment of proof-nets. The full language is more delicate to handle because the cut-rule sometimes behaves badly w.r.t. boxes; in particular, it is difficult to define a terminating normalization process for the full calculus with a Church-Rosser property. Here we have to be a little more careful:

- (i) Although the Church-Rosser property is lost, there is a semantic invariance of the normalization process which ensures that no serious divergence can be obtained.
- (ii) One can even represent a proof-net by a family of *slices*, and this representation is Church-Rosser. Unfortunately, the status of slices is far from being well-understood and secured.
- (iii) One can expect to be able to improve the syntax by reducing the number of boxes. The box of the rule (!) is an absolute one and one cannot imagine to remove it. But the boxes for \perp and ($W?$), for ($\&$) and for (\wedge) can perhaps be removed with various difficulties. The main problem is of course to find each time the ad hoc modification of the concept of trip and also to remain with a manageable condition.

Anyway, there is a strong normalization theorem for PN2 which generalizes the familiar result of [1] for F . The methodology combines the old idea of *candidat de réductibilité* (CR) with phase-like duality conditions; it is very instructive to remark that a CR defined by duality immediately has a lot of properties which, in the old works (e.g., [2]), we were forced to require explicitly!

Cut-elimination plays the role of the execution of a program; here we must recall a point that is very important: proof-nets generalize natural deduction, which in turn is isomorphic to lambda-calculi with types. Hence, proof-nets are a direct generalization of typed lambda-calculus and a proof-net is as much expressive as a functional notation. It is sometimes difficult for computer scientists to go through this point: they are shocked by the fact that PN2 does not look like a functional calculus. In fact, the earlier versions were functional, using typed linear abstraction, etc.; but the functional notation is unable to make us understand this basic fact, namely, that a (linear) term $t[x^A]$ (of type B) should be simultaneously seen as a term $u[y^{B^\perp}]$ (of type A^\perp). The functional notation is surely too expressive for our imagination to ever abolish its use; but at a very high degree of abstraction, it is clearly misleading and unadapted. But, once more, proof-nets are lambda expressions, simply written in the way best adapted to their structure. The fact that they are quite remote from our usual habits or prejudices should not be seen as an argument against their use in computer science: if there is ever any computer working

with linear logic, it is expected that the proof-nets will mostly be seen by the compiler, the user however will use more traditional devices!

III.4. Relevance for computer science

Roughly speaking, proof-nets represent programs whose termination is guaranteed by the normalization theorem. The kind of programs obtained is quite different from those coming from intuitionistic logic; let us examine this in detail in the following sections.

III.4.1. Questions and answers

In typed lambda-calculus, a term $t[x_1, \dots, x_n]$ of type $[\sigma_1, \dots, \sigma_n]\tau$ represents a *communication* between *inputs* of types $\sigma_1, \dots, \sigma_n$ and an *output* of type τ . The communication works as follows: outputs u_1, \dots, u_n of types $\sigma_1, \dots, \sigma_n$ can be substituted for x_1, \dots, x_n yielding $t[u_1, \dots, u_n]$; recall that substitution is the functional analogue of the cut-rule. Now, the terminology ‘input/output’, although correct, is a bit misleading because we have a tendency to say that t waits for the inputs u_1, \dots, u_n , whereas it is preferable to realize that the inputs are in fact the abstract symbols for inputs, i.e., the variables x_1, \dots, x_n . For that reason, the terminology *question* (for x_1, \dots, x_n) and *answer* (for t) is more suited. In other terms, x_i is a question answered by u_i when doing the substitution. The functional approach to programmation is essentially asymmetric as the communication is always between several questions and one answer: this unique answer depends on the answers to the previous questions. For instance, the term x of type σ represents a very simple type of communication: the question is to find a term u of type σ and, from an answer to this question, we get the answer corresponding to the term “ x ”, namely “ u ”. The communication is therefore just recopying.

III.4.2. Towards parallelism

Parallelism will occur as soon as we are able to break the Q/A-asymmetry between questions and answers. The most radical solution is the ability to exchange roles; i.e., a question could be seen as an answer and conversely. Lineaments of this situation appear in usual typed lambda-calculus: the term $t[x]$, which is a communication between a question of type σ and an answer of type τ , yields a term $t'[y]$ which is a communication between a question of type $\tau \Rightarrow \rho$ and an answer of type $\sigma \Rightarrow \rho$: $t'[y] = \lambda x.y(t[x])$. The roles Q/A have been exchanged during this process; however, the transformation made is not reversible: the exchange of roles is hopelessly mixed with other things and we cannot proceed any longer in this framework.

In linear logic, this limitation is overcome: questions and answers play absolutely symmetric roles, which are interchanged by *linear negation*. A proof-net with conclusions A_1, \dots, A_n can be seen as a communication net between answers of respective types $A_1^\perp, \dots, A_n^\perp$. But since we have

$$\text{answer of type } A = \text{question of type } A^\perp,$$

we can put it as, say, a communication between questions of types $A_1^\perp, \dots, A_{n-1}^\perp$ and an answer of type A_n , and this immediately sounds more familiar!

Typically, the axiom link $A \dashv A^\perp$, which is a communication between two answers of types A and A^\perp , can be seen as the trivial communication between either a question of type A^\perp and an answer of the same type, or between a question of type $A^{\perp\perp} = A$ and an answer of the type A . The cut-rule $\frac{A}{\text{CUT}} \frac{A^\perp}{A^\perp}$, which puts together two questions of orthogonal types (or a question and an answer of the same type), is the basic way to make the communication effective.

III.4.3. Communication and trips

In spite of the equality between an answer and a question of the orthogonal type, the two things should be distinguished a little; for instance, in the basic sequential case (Section III.4.1) the communication is *from* the question (variables) *to* the answer (the term). This distinction is essentially the one used for *trips*: A^\wedge stands for a question of type A^\perp , where A_\vee stands for an answer of type A . In other terms, when we move upwards, we question; when we move downwards, we answer!

The way communication works is well explained by the trips:

(i) In a CUT-link $\frac{A}{\text{CUT}} \frac{A^\perp}{A^\perp}$ the trip algorithm: ‘from A_\vee go to $A^{\perp\wedge}$ and from A_\vee go to A^\wedge ’, is easily explained as: an answer of type A is replaced by a question of type A , and an answer of type A^\perp is replaced by a question of type A^\perp : the questions met their answers.

(ii) In the axiom link $A \dashv A^\perp$, the interpretation is basically the same, the only difference being that, in the case of the cut, an *answer is changed into a question*, whereas in the case of the axiom, a *question becomes an answer*.

(iii) The multiplicatives \otimes and \wp are the two basic ways of living *concurrence*: in both cases, a question of type $(A \mathbin{m} B)^\perp$ has to be seen as two separate questions of type A^\perp and B^\perp ; an answer of type $A \mathbin{m} B$ has to be seen as two separate answers of types A and B . However,

- in the case of \otimes , there is no cooperation: if we start with A^\wedge , then we come back through A_\vee before entering B^\wedge after which we come back through B_\vee ;
- in the case of \wp , there is cooperation: if we start again with A^\wedge , then we are expected through B_\vee , from which we go to B^\wedge and eventually come back through A_\vee .

(iv) The additives $\&$ and \oplus correspond to superposition, whose typical example is the instruction IF-THEN-ELSE:

- in terms of answers, the type $A \& B$ means one answer of each type;
- in terms of questions, the type $A \& B$ means either a question of type A or a question of type B . (The meaning of \oplus is explained by duality.)

In particular, only one of the answers occurring in an answer of type $A \& B$ is of interest, but we do not know which until the corresponding question is clearly asked. This is why the rule ($\&$) works with boxes in which we put the two alternative answers so that this superposition is isolated from the rest. This is a typical lazy feature.

- (v) The exponentials ! and ? correspond to the idea of *storage*:
- an answer of type $!A$ corresponds to the idea of *writing* an answer of type A in some stable register so that it can be used *ad nauseam*.
- a question of type $?A$ corresponds to several questions of type A , 'several' being extremely unspecified (0 times: rule $(W?)$, 1 time: rule $(D?)$, etc.), i.e., is an *iterated reading*.

The fact that the rule (!) works with boxes expresses the idea of storage.

III.4.4. Work in progress

In the previous section we explained the general pattern of communication in linear logic; compared to extant work on parallelism, especially Milner's approach, the main novelty lies in the *logical approach*, i.e., parallelism should work for *internal*¹ logical reasons and not by chance. Systems like F , which are of extremely general interest for sequential computation, work because they are logically founded. (Remember that F is the system of proofs of second-order intuitionistic propositional calculus.) For similar reasons, the programs coming from linear logic are proofs within a very regular logic and this gives the good functioning of such programs (in terms of communication) at a theoretical level. At a more practical level, all these general ideas have to be put into more concrete proposals and this will be done in separate publications among which the following are expected quite soon:

- A paper with Yves Lafont, concentrating on the nonparallel aspects, in particular the new treatment of data suggested by linear logic; the formalism used is intuitionistic linear logic.
- A paper with Gianfranco Mascari, concentrating on the parallel aspects of linear logic; in particular, it will give some more details for a concrete implementation.

IV. Pons asinorum: from usual implication to linear implication²

What is the use of semantics of programming languages?

(1) Some people think that the purpose of a semantics is to interpret sets of equations in a complete way; in some sense, the semantics comes like the *official blessing* on our language, rules, etc.: they are perfect and cannot be improved. The success of this viewpoint lies in Gödel's completeness theorem which guarantees, for any consistent system of axioms, a complete semantics.

(2) A more controversial viewpoint consists in looking for *disturbing* semantics, which shed unexpected lights on the systems we know. Of course, this viewpoint is highly criticizable since there is no absolute certainty that anything will result from

¹ In this, our approach is radically different from the idea of an *external* logical comment to parallelism by means of modal, temporal, etc. logics.

² *Remark on notations:* the symbol \wp is not accessible on a word processor; I therefore propose to replace it by \sqcup and simultaneously to replace $\&$ by the symbol \sqcap ; it is important that the two symbols remind one of another.

such attempts. But the stakes are clearly higher than in the hagiographic viewpoint (1) and a change of syntax (as the one coming with linear logic) may occur from disturbing semantics.

Let us explain what is the starting disturbance that led to linear logic: the semantics of *coherent spaces* (qualitative domains in [4]), developed in Section 3 below, works as follows in the case of an implication.

IV.1. Definition. If X and Y are two coherent spaces, then a *stable function* from X to Y is a function F satisfying:

- (S1) $a \subset b \rightarrow F(a) \subset F(b)$;
- (S2) $a \cup b \in X \rightarrow F(a \cap b) = F(a) \cap F(b)$;
- (S3) F commutes with directed unions.

IV.2. Theorem (see [4, 5]). *With any stable function F from X to Y , associate the set $\text{Tr}(F)$ defined by*

$$\begin{aligned} \text{Tr}(F) = & \{(a, z); a \text{ finite object of } X, z \in F(a) \text{ such that} \\ & z \in F(b) \text{ and } b \subset a \rightarrow b = a\}. \end{aligned}$$

Then, for any $c \in X$, we have

$$F(c) = \{z; \exists a \subset c \text{ such that } (a, z) \in \text{Tr}(F)\}. \quad (1)$$

Moreover, the set of all sets $\text{Tr}(F)$, when F varies through all stable functions from X to Y , is a coherent space $X \Rightarrow Y$ which can be independently defined by

$$\begin{aligned} |X \Rightarrow Y| &= X_{\text{fin}} \times |Y| \\ (a, z) \odot (b, t) [\text{mod } X \Rightarrow Y] &\text{ iff (1)} \ a \cup b \in X \rightarrow z \odot t [\text{mod } Y], \\ &\text{(2)} \ a \cup b \in X \text{ and } a \neq b \rightarrow z \neq t. \end{aligned}$$

IV.1. Interpretation of functional languages

Theorem IV.2 is enough to interpret λ -abstraction and application in typed languages; the formation of the trace, encoding a function by a coherent set, interprets λ -abstraction whereas application is defined by means of formula (1) above. Beta- and eta-conversion are immediately seen to be sound w.r.t. this semantics.

IV.2. The disturbance

If one forgets the fact that coherent spaces are extremely simpler than Scott domains, nothing unexpected has occurred: the semantics follows the same lines as Scott-style interpretation. The novelty is that, for the first time, we have a *readable* definition of function space. The disturbance immediately comes from a close inspection of this definition: why not consider independently:

- (1) a space of repetitions $!X$,
- (2) a more restricted form of implication; the *linear* one.

IV.3. Definition. Let X and Y be coherent spaces; a stable function F from X to Y is said to be *linear* when it enjoys the two conditions:

- (L1) $a \cup b \in X \rightarrow F(a \cup b) = F(a) \cup F(b)$,
- (L2) $F(\emptyset) = \emptyset$.

Equivalently, (L1), (L2), and (S3) can be put together in one condition (L):

(L) Assume that $A \subset X$ is a set of pairwise coherent subsets of X ; then $F(\bigcup A) = \bigcup \{F(a); a \in A\}$.

IV.4. Theorem. A stable function F is linear iff its trace is formed of pairs (a, z) , where the components a are singletons.

Proof. $(a, z) \in \text{Tr}(F)$ when $z \in F(a)$: this forbids $a = \emptyset$ by (L2) and when a is minimal w.r.t. inclusion among such pairs (b, z) . Write $a = a' \cup a''$; then, from $(a, z) \in \text{Tr}(F)$, we get $z \in F(a) = F(a') \cup F(a'')$, so $z \in F(a')$ or $z \in F(a'')$. Now, the minimality of a implies that a is a singleton. \square

IV.5. Definition. For linear maps, it is more suited to forget the singleton symbols; so we get the following characterization of the space $X \multimap Y$ formed of traces of linear maps from X to Y :

$$|X \multimap Y| = |X| \times |Y|;$$

$$(x, y) \multimap (x', y') \text{ [mod } X \multimap Y] \text{ iff } \begin{aligned} (1) \quad &x \multimap x' \text{ [mod } X] \rightarrow y \multimap y' \text{ [mod } Y], \\ (2) \quad &x \multimap x' \text{ [mod } X] \rightarrow y \multimap y' \text{ [mod } Y]. \end{aligned}$$

Formula (1) is now replaced by the formula of linear application:

$$F(b) = \{y; \exists x \in b \text{ such that } (x, y) \in \text{Tr}(F)\}. \quad (2)$$

IV.6. Definition. If X is a coherent space, then $!X$ is defined as follows:

$$|!X| = X_{\text{fin}},$$

$$a \multimap b \text{ mod } !X \text{ iff } a \cup b \in X.$$

IV.3. The decomposition

We clearly have: $X = Y = !X \multimap Y$; in order to make this look like a decomposition, we have to decompose λ -abstraction and application; for linear functions, the trace and formula (2) will play the role of linear λ -abstraction and linear application. We need two other operations connected with $!$:

- (i) when $x \in X$, the formation of $!x \in !X$:

$$!x = \{a; a \in x \text{ and } a \text{ finite}\};$$

- (ii) the linearization of a function: if F is a stable map from X to Y , one can construct a linear stable map $\text{LIN}(F)$ from $!X$ to Y simply by $\text{Tr}(\text{LIN}(F)) = \text{Tr}(F)$.

Now, λ -abstraction can be viewed as two operations: first linearization, then linear λ -abstraction.

Application can be viewed as two operations too: applying a to b is the linear application of a to $!b$.

IV.4. Further questions

This already suggests to develop linear implication and $!$ for themselves. However, there are a lot of other disturbing facts; for instance, nothing prevents us from considering the result of replacing a coherent space X by its dual X^\perp where the roles of coherence and incoherence have been exchanged. But then $X \multimap Y$ is the same as $Y^\perp \multimap X^\perp$, i.e., linear functions are reversible in some sense. This immediately suggests a classical framework (symmetry between inputs/outputs, or variables/results, or questions/answers). Deep syntactic modifications come from simple semantic identifications.

1. The phase semantics

Usual finitary logics (we mean classical and intuitionistic logics) involve a certain conception of semantics which may be described as follows: truth is considered as meaningful independently of the process of verification. In particular, this imposes strong limits on the propositional parts of such logics and, for instance, starting with the general idea of *conjunction* we are hopelessly led to ‘ $A \& B$ true iff A true and B true’. By admitting several possible worlds, the intuitionistic semantics (Kripke models) allows more subtle distinctions, promoting drastically new connectives (the intuitionistic disjunction) while being extremely conservative on others (conjunction).

The change of viewpoint occurring in linear logic is simple and radical: we have to introduce an *observer*; in order to verify a *fact* A , the observer has to do something, i.e., there are tasks p, q, \dots which verify A (notation: $p /=_A A, q /=_A A, \dots$). These tasks can be seen as *phases* between a fact and its verification; phases form a monoid P and we shall consider that a fact is verified by the observer when there is no phase between him and the fact, i.e., when $1 /=_A A$. The idea of introducing phases makes a radical change on our possible connectives, which we shall explain by starting with the idea of a conjunction:

- (i) “ $\&$ ”: say that $p /=_A A \& B$ when $p /=_A A$ and $p /=_B B$;
- (ii) “ \otimes ”: say that $pq /=_A A \otimes B$ when $p /=_A A$ and $q /=_B B$.³

In the case of “ $\&$ ” the task p shares two verifications, while in the case of “ \otimes ”, the verification is done by dispatching the total task pq between A and B : p verifies A and q verifies B . These two connectives have very different behaviours, as expected from their distinct semantics.

³ The definition given here is slightly incorrect and has been simplified for pedagogic purposes.

By the way, observe that “ \otimes ” appears as a noncommutative operation. In order to keep the simplicity of the theory at this early stage of development, we shall make the hypothesis of the *commutativity* of P , i.e., we shall develop *commutative* linear logic. The noncommutative case is obviously more delicate and we have not accumulated enough materials in the commutative case to be able to foresee the general directions for a noncommutative linear logic; also we should have some real reason to shift to noncommutativity!

Semantically speaking, let us identify a fact A with the set $\{p ; p \models A\}$ so that a fact appears as a set of phases. Among all facts, we shall distinguish the *absurd* fact \perp , which is an arbitrary fixed subset of P . Given any subset G of P , we define (w.r.t. \perp) its dual G^\perp ; facts will be exactly those subsets G of P such that $G^{\perp\perp} = G$. The basic linear connectives are then easily developed as those operations leading from facts to facts.

1.1. Definition. A *phase space* P consists in the following data:

- (i) a monoid, still denoted P , and whose elements are called *phases*; the monoid is supposed to be commutative, i.e.,
 - neutrality: $1p = p1 = p$ for all $p \in P$,
 - commutativity: $pq = qp$ for all $p, q \in P$,
 - associativity: $(pq)r = p(qr)$ for all $p, q, r \in P$;
- (ii) a set \perp_P (often denoted \perp for short), the set of *antiphases* of P . (One can also say “orthogonal phases”.)

1.2. Definition. Assume that G is a subset of P ; then we define its *dual* G^\perp by

$$G^\perp = \{p \in P ; \forall q (q \in G \rightarrow pq \in \perp)\}.$$

1.3. Definition. A *fact* is a subset G of P such that $G^{\perp\perp} = G$; the elements of G are called the *phases* of G ; G is *valid* when $1 \in G$.

1.4. Immediate properties

- (i) For any $G \subset P$, $G \subset G^{\perp\perp}$;
- (ii) For any $G, H \subset P$, $G \subset H \rightarrow H^\perp \subset G^\perp$;
- (iii) G is a fact iff G is of the form H^\perp for some subset H of P .

1.5. Examples. (i) \perp is a fact because $\perp = \{1\}^\perp$.

(ii) Define a fact 1 by $1 = \perp^\perp$; then, observe that $1 \in 1$; moreover, if $p, q \in 1$, then $pq \in 1$ (more generally, if $p \in 1$, and $q \in G$, then $pq \in G^{\perp\perp}$). Hence, 1 is a submonoid of P .

(iii) Define a fact T by $T = \emptyset^\perp$; clearly, $T = P$.
 (iv) Define a fact 0 by $0 = T^\perp$; then 0 is the smallest fact (w.r.t. inclusion). Most of the time, 0 will be void.

1.6. Proposition. Facts are closed under arbitrary intersections.

Proof. If $(G_i)_{i \in I}$ is a family of facts, then $\bigcap G_i = (\bigcup G_i^\perp)^\perp$ and $\bigcap G_i$ are therefore facts. \square

1.7. Definition. The connective *nil* (*linear negation*) is defined by duality: if G is a fact, then its linear negation is G^\perp . Things have been made so that nil is an involution, i.e., $G^{\perp\perp} = G$.

1.8. Definition. The multiplicative connectives (or *multiplicatives*) are the connectives *times*, *par* and *entails*; all three are definable from the operation of product of subsets of P : if $G, H \subset P$, then $GH = \{pq ; p \in G \text{ and } q \in H\}$.

(i) If G and H are facts, their *parallelization* (connective *par*) $G \wp H$ is defined by $G \wp H = (G^\perp \cdot H^\perp)^\perp$.

(ii) If G and H are facts, their *tensorization* (connective *times*) $G \otimes H$ is defined by $G \otimes H = (G \cdot H)^\perp$.

(iii) If G and H are facts, their *linear implication* (connective *entails*) is defined by $G \multimap H = (G \cdot H^\perp)^\perp$.

1.9. Basic properties of multiplicatives. (i) Any multiplicative can be defined from any other and nil:

$$G \otimes H = (G^\perp \wp H^\perp)^\perp, \quad G \multimap H = G^\perp \wp H,$$

$$G \wp H = (G^\perp \otimes H^\perp)^\perp, \quad G \multimap H = (G \otimes H^\perp)^\perp,$$

$$G \wp H = G^\perp \multimap H, \quad G \otimes H = (G \multimap H^\perp)^\perp.$$

(ii) *par* and *times* are commutative and associative; they admit neutral elements which are \perp and 1 respectively:

$$\perp \wp G = G, \quad 1 \otimes G = G,$$

$$(G \wp H) \wp K = G \wp (H \wp K), \quad (G \otimes H) \otimes K = G \otimes (H \otimes K),$$

$$G \wp H = H \wp G, \quad G \otimes H = H \otimes G.$$

(iii) *entails* satisfies the following properties:

$$1 \multimap G = G, \quad G \multimap \perp = G^\perp,$$

$$(G \otimes H) \multimap K = G \multimap (H \multimap K), \quad G \multimap (H \wp K) = (G \multimap H) \wp K,$$

$$G \multimap H = H^\perp \multimap G^\perp.$$

Proof. (i) is immediate from the definitions; once (i) has been established, (ii) and (iii) can be reduced to the particular case of \otimes , i.e., associativity, commutativity and neutrality of 1 for this connective; among these properties only associativity deserves some attention: we shall prove that $(G \otimes H) \otimes K = (G \cdot H \cdot K)^\perp$. This property is easily reduced to the following lemma.

1.9.1. Lemma. *If F, G are two subsets of P , then $(F.G)^{\perp\perp} \supset F^{\perp\perp}.G^{\perp\perp}$.*

Proof. Assume that $p \in F^{\perp\perp}$, $q \in G^{\perp\perp}$; we have to show that, given any $v \in (F.G)^\perp$, $pqv \in \perp$; the hypothesis $v \in (F.G)^\perp$, however, means that, for all $f \in F$ and $g \in G$, $vfg \in \perp$. In particular, if we fix $f \in F$, it follows that $vf \in G^\perp = G^{\perp\perp\perp}$, hence, $vfq \in \perp$. For similar reasons, $vq \in F^\perp = F^{\perp\perp\perp}$, so $vpq \in \perp$. \square

1.10. Remark. There is the following illuminating alternative definition of $G \multimap H$: $p \in G \multimap H$ iff, for all $q \in G$, $pq \in H$.

1.11. Definition. The additive connectives (or *additives*) are the connectives *with* and *plus* which are obtained from the boolean operations:

- (i) If G and H are facts, then their *direct product* (connective *with*) $G \& H$ is defined by $G \& H = G \cap H$.
- (ii) If G and H are facts, then their *direct sum* (connective *plus*) $G \oplus H$ is defined by $G \oplus H = (G \cup H)^\perp\perp$.

1.12. Basic properties of additives. (i) *The additives are interrelated by a De Morgan principle*

$$G \& H = (G^\perp \oplus H^\perp)^\perp, \quad G \oplus H = (G^\perp \& H^\perp)^\perp.$$

(ii) *with and plus are commutative and associative; they admit neutral elements which are \top and $\mathbf{0}$ respectively:*

$$\top \& G = G,$$

$$\mathbf{0} \oplus G = G,$$

$$G \& H = H \& G,$$

$$G \oplus H = H \oplus G,$$

$$G \& (H \& K) = (G \& H) \& K,$$

$$G \oplus (H \oplus K) = (G \oplus H) \oplus K.$$

1.13. Distributivity properties. (i) \otimes is distributive w.r.t. \oplus , and \wp is distributive w.r.t. $\&$.

$$G \otimes (H \oplus K) = (G \otimes H) \oplus (G \otimes K), \quad G \wp (H \& K) \\ = (G \wp H) \& (G \wp K),$$

$$\mathbf{0} \otimes G = \mathbf{0},$$

$$\top \wp G = \top.$$

This yields, for \multimap ,

$$(G \oplus H) \multimap K = (G \multimap K) \& (H \multimap K), \quad G \multimap (H \& K) = (G \multimap H) \& (G \multimap K),$$

$$\mathbf{0} \multimap G = \top, \quad G \multimap \top = \top.$$

(ii) *Between \otimes and $\&$, between \wp and \oplus , there is only ‘half-distributivity’*

$$G \otimes (H \& K) \subset (G \otimes H) \& (G \otimes K),$$

$$(G \wp H) \oplus (G \wp K) \subset G \wp (H \oplus K)$$

which yields, in the case of \neg ,

$$(G \neg K) \oplus (H \neg K) \subset (G \& H) \neg K,$$

$$(G \neg H) \oplus (G \neg K) \subset G \neg (H \oplus K).$$

Proof. Let us concentrate on the first distributivity property; the following lemma is of general interest.

1.13.1. Lemma. *If G is any subset of P , then $G^{\perp\perp}$ is the smallest fact containing G .*

Proof. There is a smallest fact containing G , namely the intersection of all facts containing G ; let us call it S ; clearly, $S \subset G^{\perp\perp}$, from which we get $G^{\perp\perp} \subset S^{\perp\perp} \subset G^{\perp\perp\perp}$, and so $G^{\perp\perp} = S^{\perp\perp} = S$. \square

Proof of Properties 1.13 (continued). Now, observe that $G.(H \cup K)^{\perp\perp} \subset (G.(H \cup K))^{\perp}$ (Lemma 1.9.1); but $G.(H \cup K) = G.H \cup G.K$, from which we get

$$G.(H \cup K)^{\perp\perp} \subset (G.H \cup G.K)^{\perp\perp} \subset (G \otimes H) \oplus (G \otimes K).$$

By Lemma 1.13.1, $G \otimes (H \oplus K) \subset (G \otimes H) \oplus (G \otimes K)$.

Conversely, starting with $G.H \subset G \otimes (H \oplus K)$ and $G.K \subset G \otimes (H \oplus K)$, we get, by Lemma 1.13.1, $G \otimes H \cup G \otimes K \subset G \otimes (H \oplus K)$, and, by Lemma 1.13.1 once more,

$$(G \otimes H) \oplus (G \otimes K) \subset G \otimes (H \oplus K). \quad \square$$

Up to now we have described the propositional framework of linear logic. This part is the most important part of linear logic, the one which is the most radically different from usual logics. For this fragment, we shall now set up a formal system and prove a completeness theorem; afterwards, we shall have a quick overview of the modalities and quantifiers which have been deleted for reasons of clarity. These new operations, although important, will not disturb too much the ordinance of things that is now taking shape. Our first syntactical concern is economy: we essentially need a multiplicative, an additive, and nil. We get a very nice syntax by adopting the following conventions:

(i) The atomic propositions are $1, \perp, \top, 0$ together with propositional letters a, b, c, \dots and their duals $a^\perp, b^\perp, c^\perp, \dots$

(ii) Propositions are constructed from atomic ones by applying the binary connectives $\otimes, \wp, \oplus, \&$.

In particular, linear negation of a proposition is defined as follows:

$$1^\perp = \perp, \quad \perp^\perp = 1, \quad \top^\perp = 0, \quad 0^\perp = \top,$$

$$a^{\perp\perp} = a \quad (a^\perp \text{ is already in the syntax}),$$

$$(A \otimes B)^\perp = A^\perp \wp B^\perp, \quad (A \wp B)^\perp = A^\perp \otimes B^\perp,$$

$$(A \oplus B)^\perp = A^\perp \& B^\perp, \quad (A \& B)^\perp = A^\perp \oplus B^\perp.$$

This particular form of syntax without negation is not something deep; once more, this is just a way to avoid repetitions.

1.14. Definition. (i) A *phase* structure for the propositional language just described consists in a phase space (P, \perp_P) and, for each propositional letter a , a fact a_S of P .

(ii) With each proposition we associate its interpretation, i.e., a fact of P , in a completely straightforward way: the interpretation of A in the structure S is denoted by A_S or $S(A)$.

(iii) A is valid in S when $1 \in S(A)$.

(iv) A is a *linear tautology* when A is valid in any phase structure S .

1.15. Definition. We develop here a sequent calculus for linear logic; as expected from the peculiarities of our syntax, the sequents will have no left-hand side, i.e., they will be of the form $\vdash A_1, \dots, A_n$. The real change w.r.t. usual logics consists in *the dumping of structural rules* and, in fact, only the rule of exchange is left. Such a situation is more familiar than it seems at first sight since intuitionistic logic is built on such a restriction: the unique right-hand side formula in a special place in the sequent where contraction is forbidden. In linear logic this interdiction is extended to the whole sequent and weakening is forbidden as well.

Logical axioms:

$$\vdash A, A^\perp.$$

(It is enough to restrict it to the case of a propositional letter a .)

Cut rule:

$$\frac{\vdash A, B \quad \vdash A^\perp, C}{\vdash B, C} \text{CUT.}$$

Exchange rule:

$$\frac{\vdash A}{\vdash B} \text{EXCH,}$$

where B is obtained by permuting the formulas of A .

Additive rules:

$$\vdash T, A \quad (\text{axiom}, A \text{ arbitrary}) \quad (\text{no rule for } 0),$$

$$\frac{\vdash A, C \quad \vdash B, C}{\vdash A \& B, C} \&, \quad \frac{\vdash A, C}{\vdash A \oplus B, C} 1 \oplus \frac{\vdash B, C}{\vdash A \oplus B, C} 2 \oplus.$$

Multiplicative rules:

$$\vdash 1 \quad (\text{axiom}), \quad \frac{\vdash A}{\vdash \perp, A} \perp,$$

$$\frac{\vdash A, C \quad \vdash B, D}{\vdash A \otimes B, C, D} \otimes, \quad \frac{\vdash A, B, C}{\vdash A \wp B, C} \wp.$$

The additional possibilities of linear logic can be ascribed to the dumping of the structural rules: in presence of these rules, for instance the rule $(\&)$ and the rule (\otimes) would amount to the same. Without structural rules, the *contexts* (i.e., the unspecified lists of formulas A, B, C , etc.) which occur in the rules take a greater importance. In particular, if we start with the idea of a conjunction, there are two reasonable ways to handle the contexts: by juxtaposition (connective \otimes) or by identification (connective $\&$). By the way, observe that contexts behave like phases and this will be the basis of the completeness argument.

1.16. Proposition. *Linear sequent calculus is sound w.r.t. validity in phase structures.*

Proof. We have not defined the interpretation of a sequent $\vdash A$; but, obviously, $\vdash A$ must be read as the *par* of its components (and, when the list is void, as \perp). We prove that, whenever $\vdash A$ is provable, then it is valid in any phase structure $(P, \perp_P, (a_S))$. We argue by induction on a proof of A :

- (i) The proof consists of the axiom $\vdash B, B^\perp. B \otimes B^\perp = B \multimap B$ and, by Remark 1.10, $1 \in S(B \multimap B) = S(B) \multimap S(B)$.
- (ii) The proof consists of the axiom $\vdash T, A$. The interpretation is of the form $S(T) \otimes G$, with G as fact; equivalently, $S(0) \multimap G$, but since $S(0)$ is the smallest fact of P , 0_P is included in G and, by Remark 1.10, $1 \in S(0)$.
- (iii) The proof consists of the axiom $\vdash 1$. The interpretation of 1 is 1_P and we have already observed that $1 \in 1_P$ (Example 1.5(ii)).
- (iv) The proof ends with a cut-rule. To prove the soundness of this rule, we have to show that, in case $1 \in G \otimes H$ and $1 \in G^\perp \otimes K$, then $1 \in H \otimes K$. This is easily seen if one puts $G \otimes H$, $G^\perp \otimes K$, and $H \otimes K$ under the forms $H^\perp \multimap G$, $G \multimap K$, and $H^\perp \multimap K$ and if one then applies Remark 1.10.
- (v) The proof ends with an exchange rule: immediate.
- (vi) The proof ends with $(\&)$. In order to prove the soundness of $(\&)$, we have to show that, in case $1 \in G \otimes K$ and $1 \in H \multimap K$, then $1 \in (G \& H) \multimap K$. This is immediate from the distributivity law 1.13(i).
- (vii) The proof ends with $(1\oplus)$ or $(2\oplus)$. In order to prove the soundness of these rules, we have for instance to show that, in case $1 \in G \otimes K$, then $1 \in (G \oplus H) \otimes K$, which is immediate from the half-distributivity of *par* w.r.t. *plus* (c.f. Property 1.13(ii)).
- (viii) The proof ends with (\perp) . In order to prove the soundness of (\perp) , we have to show that, in case $1 \in G$, then $1 \in S(\perp) \otimes G$; this is immediate from $\perp_P \multimap G = G$.
- (ix) The proof ends with (\otimes) . In order to prove the soundness of (\otimes) , we have to show that, in case $1 \in F \otimes G$, $1 \in H \otimes K$, then $1 \in (F \otimes H) \otimes G \otimes K$. If these three facts are put as $G^\perp \multimap F$, $K^\perp \multimap H$, and $(G^\perp \otimes K^\perp) \multimap (F \otimes H)$, then we have, by hypothesis, $G^\perp \subset F$, $K^\perp \subset H$ from which one deduces $G^\perp \otimes K^\perp \subset F \otimes H$.
- (x) The proof ends with (\otimes) : immediate. \square

1.17. Theorem. *Linear sequent calculus is complete w.r.t. phase semantics.*

Proof. For the time of the proof, let us adopt the following convention: we are dealing with sequences of formulas A, B, C, \dots up to their order, i.e., we are dealing with *multisets* of formulas. Multisets of formulas form a commutative monoid, when equipped with concatenation $*$ and with \emptyset as a neutral element. This defines a phase space M , and the antiphases of M will just be those multisets A such that $\vdash A$ is provable in linear sequent calculus (pedantically, in the variant without exchange, adapted to multisets of formulas). We claim the existence of a phase structure $(M, \perp_M, (a_S))$ such that, for all formulas A ,

$$S(A) = \{B; \vdash A, B \text{ is provable in linear sequent calculus}\}. \quad (3)$$

Let us call the right-hand side of the equation (3) $\Pr(A)$.

A crucial observation is that sets $\Pr(A)$ are facts of M ; more precisely, that $\Pr(A) = \Pr(A^\perp)^\perp$: if $B \in \Pr(A)$ and if $C \in \Pr(A^\perp)$, then $\vdash A, B$ and $\vdash A^\perp, C$ are both provable and, by CUT, so is $\vdash B, C$; hence, $B * C \in \perp_M$. This shows that $\Pr(A) \subset \Pr(A^\perp)^\perp$.

Conversely, if $B \in \Pr(A^\perp)^\perp$, observe that since $\vdash A, A^\perp$ is an axiom, $A \in \Pr(A^\perp)$; hence, $A * B \in \perp_M$, i.e., $\vdash A, B$ is provable; but then $B \in \Pr(A)$. It is therefore legitimate to define a phase structure S by setting $S(a) = \Pr(a)$ for any propositional letter a .

By induction on the formula A , we prove that $S(A) = \Pr(A)$:

- (i) A is an atom a : by definition.
- (ii) A is of the form a^\perp : because $S(a^\perp) = S(a)^\perp = \Pr(a)^\perp = \Pr(a^\perp)$.
- (iii) A is \perp : observe that $\perp_M = \Pr(\perp)$ (immediate).
- (iv) A is 1 : as in (ii).
- (v) A is T : the axiom for T yields $S(T) = M = \top_M$.
- (vi) A is 0 : as in (ii).
- (vii) A is $B \& C$: it is easily seen that $\vdash B \& C, D$ is provable iff $\vdash B, D$ and $\vdash C, D$ are both provable. From this we get $\Pr(A \& B) = \Pr(A) \& \Pr(B)$ which is what is needed to go through this step.
- (viii) A is $B \oplus C$: from $\Pr(A \& B) = \Pr(A) \& \Pr(B)$, we easily obtain $\Pr(A \oplus B) = \Pr(A) \oplus \Pr(B)$, etc.
- (ix) A is $B \otimes C$: $\Pr(A) \cdot \Pr(B) \subset \Pr(A \otimes B)$ (by the \otimes -rule); hence, by Lemma 1.13.1, $\Pr(A) \otimes \Pr(B) \subset \Pr(A \otimes B)$. Conversely, if $C \in \Pr(A \otimes B)$ and $D \in (\Pr(A) \cdot \Pr(B))^\perp$, then $\vdash D, A^\perp, B^\perp$ is provable and, by the rule (\wp) and a cut, we get a proof of $\vdash D, C$. This shows the reverse inclusion, namely, $\Pr(A \otimes B) \subset \Pr(A) \otimes \Pr(B)$. From this we can easily conclude our statement.
- (x) A is $B \wp C$: similar to (viii).

Now, assume that A is a linear tautology; then A is valid in S , which means that $\emptyset \in S(A) = \Pr(A)$; but then $\vdash A$ is provable in linear sequent calculus. \square

1.18. Remarks. (i) Usually, a completeness is stated, not for logical calculi, but for theories. We have given absolutely no meaning to the concept of ‘linear logical theory’. The reason is that if we were allowing an axiom, say A , to say that B is provable with the help of A , this has nothing to do with the provability of $A \multimap B$;

in fact, this is equivalent to the provability of $!A \multimap B$, where “!” is the connective ‘of course’, which has not yet been introduced.

(ii) The phase spaces P have been supposed to be commutative. If P were not commutative and if we restrict our attention to *distinguished* subsets of P (i.e., $p.G = G.p$ for all $p \in P$) (in particular, \perp_P must be distinguished), then one can develop the same theory as we did, and the connectives \otimes and \wp are still commutative. Of course, when we were thinking of a possible ‘noncommutative linear logic’, this is not the straightforward generalization that we have in mind, but something for which the multiplicatives would be noncommutative.

Now that the main features of linear logic have been disclosed, let us complete the language to obtain a predicate calculus with the same expressive power as the usual ones. The first effort has to be made on the propositional part: one must be able to recover somewhere the structural rules. Let us call $?A$ a formula obtained from A which is ‘ A -saturated w.r.t. the structural rules’. The fact that $?A$ comes from A can be expressed as $A \subset ?A$; the fact that one can weaken on $?A$ can be expressed as $\perp \subset ?A$, and the fact that one can contract on $?A$ as $?A \wp ?A \subset ?A$. But there are also other properties of $?A$ and everything will be summarized in the following definition.

1.19. Definition. A *topolinear space* (P, \perp, \mathbb{F}) consists in a phase space (P, \perp) together with a set \mathbb{F} of facts of (P, \perp) , the *closed facts*; the following conditions are required:

- (i) \mathbb{F} is closed under arbitrary *with*; in particular, $P \in \mathbb{F}$.
- (ii) \mathbb{F} is closed under finite *par*; in particular, $\perp \in \mathbb{F}$.
- (iii) \perp is indeed the smallest fact of \mathbb{F} : $\perp \subset F$ for all $F \in \mathbb{F}$.
- (iv) *par* is idempotent on \mathbb{F} : $F \wp F = F$ for all $F \in \mathbb{F}$.

The linear negations of closed facts are called *open facts*.

1.20. Definition. The exponential connectives or *modalities* are the connectives *of course* and *why not*; both of them are definable in topolinear spaces:

- (i) If G is a fact, then its *affirmation* (connective *of course*) $!G$ is defined as the greatest (w.r.t. inclusion) open fact included in G (the *interior* of G).
- (ii) If G is a fact, then its *consideration* (connective *why not*) $?G$ is defined as the smallest (w.r.t. inclusion) closed fact containing G (the *closure* of G).

We now adapt our syntax so that we can handle the two modalities; first, the *definition* of linear negation is extended to the modalities by

$$(!A)^\perp = ?(A^\perp), \quad (?A)^\perp = !(A^\perp).$$

Then we have to give sequent rules for the modalities. In the definition below, $?C$ is short for a sequence $?C_1, \dots, ?C_n$.

1.21. Definition. Linear sequent calculus is enriched with the following rules for modalities:

Exponential rules:

$$\frac{\vdash A, B}{\vdash ?A, B} D? \quad (\text{dereliction}),$$

$$\frac{\vdash B}{\vdash ?A; B} W? \quad (\text{weakening}), \quad \frac{\vdash A, ?B}{\vdash !A, ?B} !,$$

$$\frac{\vdash ?A, ?A, B}{\vdash ?A, B} C? \quad (\text{contraction}).$$

1.22. Theorem. *The calculus as extended in Definition 1.21 is sound and complete w.r.t. validity in topolinear structures, i.e., the obvious analogue of phase structures where phase spaces have been replaced by topolinear spaces.*

Proof. (i): *soundness.* The soundness of dereliction comes from $F \subset ?F$. The soundness of weakening comes from $\perp \subset ?F$ and the soundness of contraction from $?F = ?F \wp ?F$. The soundness of (!) is the fact that if $F \subset ?G_1 \wp \dots \wp ?G_n$, then $?F \subset ?G_1 \wp \dots \wp ?G_n$. This is because $?G_1 \wp \dots \wp ?G_n$ is a closed fact and because $?F$ is the smallest closed fact containing F .

(ii): *completeness.* We define the phase space M as in the proof of Theorem 1.17. Now we define a topolinear structure \mathbb{F} on M by saying that the closed facts are just arbitrary intersections (i.e., *with*) of facts of the form $S(?A)$.

1.22.1. Lemma. *The set \mathbb{E} of all facts of the form $S(?A)$ satisfies properties (ii)–(iv) (Definition 1.19) of topolinear spaces.*

Proof. (ii): The axiomatic system just described proves the linear equivalence $?(\mathbf{A} \oplus \mathbf{B}) \rightsquigarrow (?A) \wp (?B)$ ($C \rightsquigarrow D$ is short for $(C \multimap D) \& (D \multimap C)$):

$$\begin{array}{cccc} \frac{\vdash A^\perp, A}{\vdash A^\perp, A \oplus B} 1 \oplus & \frac{\vdash B^\perp, B}{\vdash B^\perp, A \oplus B} 2 \oplus & \frac{\vdash A^\perp, A}{\vdash A^\perp, ?A} D? & \frac{\vdash B^\perp, B}{\vdash B^\perp, ?B} D? \\ \frac{\vdash A^\perp, A \oplus B}{\vdash !A^\perp, ?A \oplus B} D? & \frac{\vdash B^\perp, A \oplus B}{\vdash !B^\perp, ?A \oplus B} D? & \frac{\vdash A^\perp, ?A}{\vdash A^\perp, ?A, ?B} W? & \frac{\vdash B^\perp, ?B}{\vdash B^\perp, ?A, ?B} W? \\ \frac{\vdash A^\perp, ?A \oplus B}{\vdash !A^\perp, ?A \oplus B} ! & \frac{\vdash B^\perp, ?A \oplus B}{\vdash !B^\perp, ?A \oplus B} ! & \frac{\vdash A^\perp, ?A, ?B}{\vdash A^\perp \& B^\perp, ?A, ?B} \& & \frac{\vdash B^\perp, ?A, ?B}{\vdash B^\perp \& B^\perp, ?A, ?B} \& \\ \frac{\vdash !A^\perp, ?A \oplus B \vdash !B^\perp, ?A \oplus B}{\vdash !A^\perp \otimes !B^\perp, ?A \oplus B} \otimes & & \frac{\vdash A^\perp \& B^\perp, ?A, ?B}{\vdash !A^\perp \& B^\perp, ?A, ?B} ! & \\ \frac{\vdash !A^\perp \otimes !B^\perp, ?A \oplus B}{\vdash !A^\perp \otimes !B^\perp, ?A \oplus B} \wp & \frac{\vdash !A^\perp \& B^\perp, ?A, ?B}{\vdash !A^\perp \& B^\perp, ?A \wp ?B} \wp & & \\ \frac{\vdash !(?A \wp ?B) \multimap ?(\mathbf{A} \oplus \mathbf{B})}{\vdash !(?A \wp ?B) \rightsquigarrow ?(\mathbf{A} \oplus \mathbf{B})} & \frac{\vdash !(?A \wp ?B)}{\vdash !(?A \wp ?B) \multimap ?(\mathbf{A} \oplus \mathbf{B})} & & \\ & & & \frac{\vdash !(?A \wp ?B) \rightsquigarrow ?(\mathbf{A} \oplus \mathbf{B})}{\vdash !(?A \wp ?B) \rightsquigarrow ?(\mathbf{A} \oplus \mathbf{B})} \& \end{array}$$

From this linear equivalence, it is plain that $S(?A) \wp S(?B) = S(?A \wp ?B) = S(?(\mathbf{A} \oplus \mathbf{B}))$, so \mathbb{E} is closed by *par*.

(iii): The equivalence $\perp \rightsquigarrow ?0$ is easily proved. From this, $\perp \in E$; also the weakening rule shows that $M(\perp) \subset M(?A)$.

(iv): From the equivalence $A \rightsquigarrow A \oplus A$, it is easy to obtain $?A \rightsquigarrow ?(A \oplus A)$, and so $?A \rightsquigarrow (?A \wp ?A)$. From this we get $S(?A) = S(?A) \wp S(?A)$. \square

Proof of Theorem 1.22 (continued). Now, F is made of arbitrary intersections of elements of E . From this it is immediate that F satisfies properties (i) and (iii) of toplinear spaces.

Now, if we observe that \wp is distributive w.r.t. arbitrary intersections, it is not difficult to get (ii) and (iv) for F from Lemma 1.22.1. For instance,

$$(\bigcap S(?A_i)) \wp (\bigcap S(?A_i)) = \bigcap S(?A_i) \wp S(?A_i)$$

because $S(?A_i) \wp S(?A_j) \supset S(?A_i)$, which in turn follows from $S(?A_j) \supset \perp$, etc.

We have established that F is a toplinear structure. Now, F is defined in such a way that, in F ,

$$?S(A) = \bigcap \{S(?B); S(A) \subset S(?B)\} = \bigcap \{S(?B); \vdash A \multimap ?B \text{ is provable}\}.$$

But, by rule (!), $\vdash A \multimap ?B$ is provable iff $\vdash ?A \multimap ?B$ is; i.e., we finally find out that $?S(A) = S(?A)$ and this is enough to finish the completeness argument for Theorem 1.22. \square

1.23. Remark. In spite of the obvious analogies with the modalities of modal logic, it is better to keep distinct notations, for at least two reasons:

(ii) ! and ? are modalities within linear logic, while \Box and \Diamond are modalities inside usual predicate calculus; in particular, the rule (!) which looks like the introduction rule of \Box is perhaps not so close since the sequents are handled in a completely different way.

(ii) The adoption of the modal symbolism would convey the idea of ‘yet another modal system’, which is not quite the point of linear logic

1.24. Definition. The *quantifiers* are semantically defined as infinite generalizations of the additives; these quantifiers are \wedge (*any*) and \vee (*some*). The semantic definitions are straightforward and boring and are left to the reader. The syntactic rules are *Quantifier rules*:

$$\frac{\vdash A, B}{\vdash \wedge x A, B} \wedge, \quad \frac{\vdash A[t/x], B}{\vdash \vee x A, B} \vee.$$

(\wedge) is subject to the familiar restriction on variables: x not free in B . If the reader has written the boring definition of the semantics, he can in turn prove the following boring theorem.

1.25. Theorem. *Phase semantics is sound and complete w.r.t. linear predicate calculus.*

1.26. Remark. \wedge and \vee are De-Morgan dual; \vee is effective, which does not prevent it from being a linear negation!

2. Proof-nets

To understand what is going on in this section, think of the two formalisms for the intuitionistic fragment ($\wedge, \rightarrow, \forall$): NJ (natural deduction) and LJ (sequent calculus). In LJ, the portion of proof from $A, C, E \vdash F$ to $A \wedge B, C \wedge D, E \vdash F$ can be written in two different ways, whereas in NJ there is only one way. According to Prawitz, the proofs in NJ should be viewed as true ones, while those in LJ are no longer primitive, i.e., a proof in LJ gives us instructions enabling us to recover a real proof in NJ. The issue is important, especially if we think of equality of normal forms and related questions. Unfortunately, already in the case of intuitionistic logic, something goes wrong with NJ when we consider the full language: the proofs in NJ cannot any longer be viewed as primitive since one needs additional identifications between them (*commutation rules*). The attempts to fix these defects by means of ‘multiple conclusion logics’ were never convincing.

Linear logic is faced with the same question with a greater acuteness: on the one hand, since linear logic is constructive, linear proofs must be seen as programs whose execution involves a normal-form theorem. On the other hand, linear logic is ‘classical’ to some extent, which means that the commutation problems are bigger than they are in the intuitionistic case. These constraints made it necessary to have a second look at this question of ‘multiple conclusion logic’, but now making use of the linear connectives which enable us to make distinctions that were not accessible to people working on the same subject with usual logics.

Proof-nets, which we are going to develop in this section, are the result of these investigations. The core of the theory is the multiplicative fragment, which works in an extremely satisfactory way. The extension of the theory to the full language is not so good, but good enough to keep a reasonable fragment of the calculus free from any criticism. We first start with the *multiplicative fragment*: by this we mean the formulas built from atoms $a, b, \dots, a^\perp, b^\perp$, by means of the connectives \otimes and \wp . There are three basic ingredients that will be used to produce proof-nets.

(i) Axioms:

$$\frac{}{A \quad A^\perp},$$

(ii) \otimes -rules:

$$\frac{A \quad B}{A \otimes B},$$

(iii) \wp -rules:

$$\frac{A \quad B}{A \wp B}.$$

(The cut-rule, which has exactly the same geometrical structure as the \otimes -rule is omitted from this study.)

Using these three patterns, we can construct (in a way not difficult to imagine) *proof-structures* with several conclusions A_1, \dots, A_n . We would like to make sure that such structures are sound, i.e., that they prove the *par* of their conclusion, i.e., $A_1 \wp \dots \wp A_n$. With usual proofs (written as trees), the soundness condition is local (i.e., one checks each rule separately), while, in a pattern with several conclusions, the left-hand side cannot act in contradiction with the right-hand side as typified in Fig. 2. Fig. 2(a) is sound, since $\vdash A \otimes B, A^\perp \wp B^\perp$ is provable, but Fig. 2(b) is not. In fact, (a) is sound because the two rules used are distinct. But what is the general global criterion of correctness?



Fig. 2.

2.1. Definition.

A *proof-structure* consists of the following objects:

(i) Occurrences of formulas (i.e., to be pedantic, pairs (A, i) , where A is the formula and the second component i serves to distinguish between two occurrences, etc.); there must be at least one formula occurring.

(ii) A certain number of *links* between these (occurrences of) formulas. These links, which must be viewed as binary or ternary relations, are of the three kinds already mentioned:

Axiom link: $\text{AX}(A, i; A^\perp, j)$ between an occurrence of A and an occurrence of A^\perp . This link is considered to be symmetric, i.e., it is the same as $\text{AX}(A^\perp, j; A, i)$. This link has no *premise* and two *conclusions*, namely (A, i) and (A^\perp, j) .

Times link: $\otimes(A, i; B, j; A \otimes B, k)$. This link is not the same as $\otimes(B, j; A, i; A \otimes B, k)$; i.e., the link is not symmetric. The *premises* of the link are (A, i) and (B, j) , and its *conclusion* is $(A \otimes B, k)$.

Par link: $\wp(A, i; B, j; A \wp B, k)$, whose *premises* are (A, i) and (B, j) and whose *conclusion* is $(A \wp B, k)$. This link is not symmetric.

- (iii) We require that
 - every occurrence of a formula in the structure is the conclusion of one and only one link;
 - every occurrence of a formula in the structure is the premise of at most one link.

2.2. Examples.

What we have pedantically defined as a proof-structure corresponds to the rough idea of starting with the axiom links and then developing the proof in a tree-like way by using the \otimes - and \wp -rules. In particular, Fig. 2(a) and (b) are equally satisfactory as proof-structures. One may imagine more pathological

examples, e.g., nonconnected proof-structures:

$$\overline{A} \quad A^\perp \quad \overline{B^\perp} \quad B.$$

(In the sequel, proof-nets will be connected.)

We shall content ourselves with a graphical representation for proof-structures of the kind already used. Such a representation is rarely ambiguous and so easy to understand. Let us however recall that in an axiom link, the left formula is here for convenience (symmetry of the link), while in the two other types of link, left and right have a real meaning. We shall not define obvious notions such as being above/below a formula, or being connected. By the way, we have already started with our last abuse: we speak of a formula instead of an occurrence, any time this is not ambiguous.

2.1. The concept of trip

The problem is to determine which among the proof-structures are logically sound; such structures will be called *nets*. For this we shall define the concept of *trip*, which is better understood within a temporal imagery and the idea of something (a particle, information, etc.) travelling through the proof-structure.

(i) The ‘time’ will be *cyclic*, discrete and finite. In general, the trips will be scheduled according to some Z/kZ , but the choice of the origin of time will be just a convention with no meaning. The problem with cyclic time is that ‘after’ and ‘before’ are not defined globally. However, if t and u are two distinct times, the interval $[t, u]$ makes sense: choose a determination r between 0 and $k - 1$ for $u - t$; then, $[t, u]$ consists of the time moments $t, t + 1, \dots, t + r = u$.

(ii) Each formula will be viewed as a box (see Fig. 3) in which our particle may travel. The idea is that the particle will enter A by the gate marked “ \blacktriangleleft ” and will

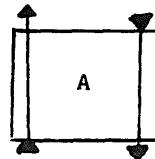


Fig. 3.

immediately go out through the gate marked “ \triangleright ”. These two operations are performed in the same unit of time t^\wedge ; t^\wedge is used only for this purpose; i.e., at t^\wedge the particle is in A between “ \blacktriangleleft ” and “ \triangleright ” and nowhere else. The particle will re-enter (but there is no ‘before’, no ‘after’) A by “ \lhd ” and will immediately go out through “ \downarrow ”, all this at another moment t_\vee . Now the particle exits through a gate with a direction; at the moment just after this exit, it will enter another formula by using the links of the proof-structure. Sometimes, the trip can only be done in one way, but very often, there are several ways of continuing, and these ways are selected by *switches*.

(1) *Axiom link*: The picture in Fig. 4 is clear: immediately after exiting through gate A^\uparrow , the particle enters $A^\perp\Downarrow$; immediately after exiting through gate $A^\perp\uparrow$ the particle enters $A\Downarrow$. In other terms, $t(A^\perp) = t(A^\wedge) + 1$ and $t(A_\vee) = t(A^{\perp\wedge}) + 1$.

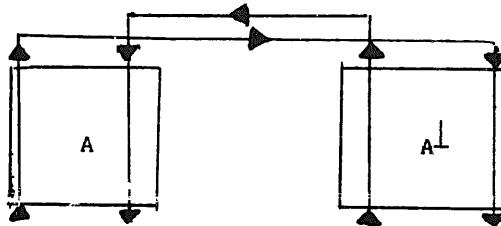


Fig. 4.

(2) *Terminal formula*: This is the case of a formula which is not a premise (see Fig. 5), i.e., just after exiting through $A\downarrow$, the particle re-enters through $A\downarrow$. In other terms, $t(A^\wedge) = t(A_\vee) + 1$.

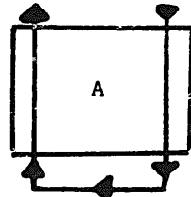


Fig. 5.

(3) *Times link*: Associated with such a link is a *switch* with two positions “L” and “R”. The switch is set on one of these two positions for the whole trip, independently of the positions of other switches. According to the position of the switch, the particle moves as shown in Fig. 6.

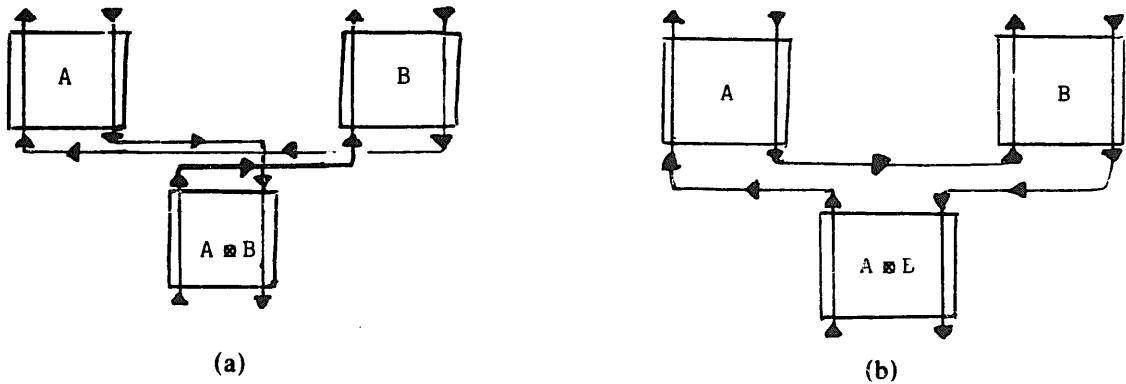


Fig. 6. (a) switch on “L”. (b) switch on “R”.

Looking carefully, we see that the only difference between “L” and “R” is the interchange between A and B . The terminology “L” comes from the fact that we reach the conclusion $(A \otimes B)^\Downarrow$ through the left premise $(A\downarrow)$ when the switch is on

“L”. The same reason is behind the terminology for the *par* link.

$$\text{“L”}: \quad t(B^\wedge) = t(A \otimes B^\wedge) + 1, \quad t(A^\wedge) = t(B_v) + 1, \quad t(A \otimes B_v) = t(A_v) + 1;$$

$$\text{“R”}: \quad t(A^\wedge) = t(A \otimes B^\wedge) + 1, \quad t(B^\wedge) = t(A_v) + 1, \quad t(A \otimes B_v) = t(B_v) + 1.$$

(4) *Par link* (see Fig. 7): Associated with such a link is a switch with two positions “L” and “R”; here too, the switch is set once and for all for the trip, independently

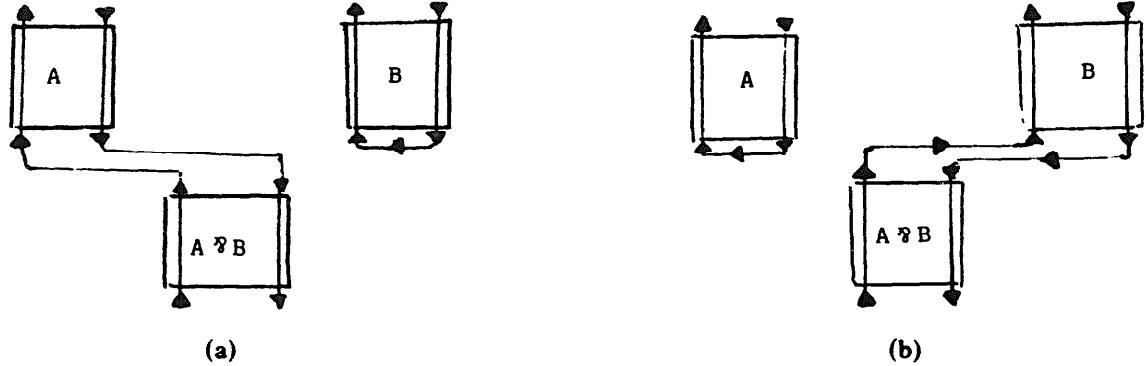


Fig. 7. (a) switch on “L”. (b) switch on “R”.

from other switches. The two positions of the switch lead to symmetric travels of the particle.

$$\text{“L”}: \quad t(A^\wedge) = t(A \wp B)^\wedge + 1, \quad t(A \wp B_v) = t(A_v) + 1, \quad t(B^\wedge) = t(B_v) + 1;$$

$$\text{“R”}: \quad t(B^\wedge) = t(A \wp B)^\wedge + 1, \quad t(A \wp B_v) = t(A_v) + 1, \quad t(A^\wedge) = t(A_v) + 1.$$

(iii) Everything is now ready for the trip, which can be made as follows: we set the switches of all *par* and *times* links on arbitrary positions (so there are 2^n possibilities if n is the number of switches). We select an arbitrary formula A and an exit gate ($A\uparrow$ or $A\downarrow$) at time 0. Then there are clear, unambiguous conditions to go on forever. Since the whole structure is finite, the trip is eventually periodic, but since the same remark can be made if one inverts the sense of time, this concretely forces the existence of a strictly positive integer k such that, at time k , the particle enters through the dual gate ($A\downarrow$ or $A\uparrow$ respectively); the number k with this property can be chosen minimal and then we have obtained a *trip* parameterized by Z/kZ . Now, two possibilities arise:

- (a) $k < 2p$, where p is the number of formulas of the structure; the trip is called a *shorttrip*.
- (b) $k = 2p$, with p as above; the trip is called a *longtrip*.

2.3. Examples. Go back to the examples in Fig. 2 and, in both cases, set all switches on “L”: in Fig. 2(a) we get a longtrip, namely,

$$A^\wedge, A_v^\perp, A^\perp \wp B_v^\perp, A^\perp \wp B^{\perp\wedge}, A^{\perp\wedge}, A_v, A \otimes B_v, A \otimes B^\wedge, B^\wedge, B_v^\perp, B^{\perp\wedge}, B_v, A^\wedge, \dots$$

Here, each formula has been passed in both senses. In Fig. 2(b), however, we get

$$A^\wedge, A_v^\perp, A^\perp \otimes B_v^\perp, A^\perp \otimes B^{\perp\wedge}, B^{\perp\wedge}, B_v, A^\wedge, \dots,$$

a typical shorttrip. These examples are enough to give some evidence for the following definition.

2.4. Definition. A proof-structure is said to be a *proof-net* when it admits no shorttrip.

2.5. Reformulation. We reformulate the definition of proof-net in more usual terms: A proof structure with p formulas and n switches is said to be a *proof-net* when, for any position of the switches, there is a bijection $t(\cdot)$ between $Z/2pZ$ and the $2p$ distinct exits of the structure such that, for any two exits e, e' , $t(e') = t(e) + 1$ iff e' comes immediately after e in the travel process that has been exposed in full details in the introduction of Section 2.1 (w.r.t. the positions of the switches, of course).

2.6. Remark. Checking that a proof-structure has no shorttrip requires looking at 2^n different cases. This number can be decreased to 2^{n-1} : let (1) and (2) be two positions of the switches, (2) being obtained from (1) by commuting all \otimes -switches to the other position. It is easily checked that (2) is the same as (1) *but with time reversed* and, in particular, it suffices to check only one of them. Anyway, the soundness condition is not feasible. However, it is not part of our intentions to *check* soundness by concrete means. The proof-nets we shall deal with in practice will all come from sequent calculus or will be obtained from other proof-nets by means of transitions preserving the soundness condition. Hence, the soundness condition is an abstract notion (just like, say, semantic soundness), whose importance lies in its relation to linear sequent calculus. We shall now prove that proof-nets are the ‘natural deduction of linear sequent calculus’.

2.7. Theorem. If π is a proof in linear sequent calculus of $\vdash A_1, \dots, A_n$ (in fact, in the multiplicative fragment without cut), then we can naturally associate with π a proof-net π^- whose terminal formulas are exactly (one occurrence of) $A_1, \dots, (one occurrence of) A_n$.

Proof. The proof-net π^- is defined by induction on π as follows.

Case 1: π is an axiom $\vdash A, A^\perp$; obviously, one must define π^- as

$$\overline{A \quad A^\perp}$$

which is a proof-net: there is no switch, and only one longtrip $A^\wedge, A_v^\perp, A^\perp \wedge, A_v, A^\wedge, \dots$

Case 2: π is obtained from λ by an exchange rule; take $\pi^- = \lambda^-$.

Case 3: π is obtained from λ by (\wp):

$$\frac{\overset{\lambda}{\vdash A, B, C}}{\vdash A \wp B, C}.$$

By induction hypothesis, we have obtained λ^- , in which we can individualize A and B and π^- is obtained as follows

$$\frac{\overset{\lambda^-}{A \quad B}}{A \wp B}.$$

We have to prove the soundness of π^- : Setting all switches on arbitrary positions in π^- and assuming that the switch of the new link is on, say “L”, we can start a trip at $t=0$ and A^L ; since λ^- is sound, at time $2n-1$ (where n is the number of formulas of λ^-) we arrive at A_v ; then the trip is easily finished by adding $A \wp B_v, A \wp B^L$ at times $2n, 2n+1$. We have obtained a longtrip, so π^- is correct.

Case 4: π is obtained from λ and μ by (\otimes):

$$\frac{\overset{\lambda}{\vdash A, C} \quad \overset{\mu}{\vdash B, D}}{\vdash A \otimes B, C, D}.$$

By induction hypothesis, we have obtained λ^- and μ^- , in which we can respectively individualize A and B . π^- is obtained as follows:

$$\frac{\overset{\lambda^-}{A} \quad \overset{\mu^-}{B}}{A \otimes B}.$$

Again, set all switches of π^- for a trip, and let us say that the last switch is on “R”. Assume that there are n formulas in λ^- , m formulas in μ^- : starting with A^L at time $t=0$, we arrive at A_v at time $t=2n-1$ (soundness of λ^-); then, at time $t=2n$, we move at B^L and so, since μ^- is sound, we arrive at B_v at time $t=2n+2m-1$. Then, at times $2n+2m$ and $2n+2m+1$, we visit $A \otimes B_v$ and $A \otimes B^L$, therefore ending a longtrip. Hence, π^- is sound. \square

2.8. Remark. The transformation $\pi \rightsquigarrow \pi^-$ identifies proofs which differ by the order of rules. For instance, the proofs

$$\begin{array}{c} \vdash A, A^\perp \quad \vdash B, B^\perp \\ \hline \vdash A \otimes B, A^\perp, B^\perp \\ \hline \vdash A \otimes B, A^\perp \wp B^\perp \quad \vdash C, C^\perp \\ \hline \vdash (A \otimes B) \otimes C, A^\perp \wp B^\perp, C^\perp \end{array}$$

$$\begin{array}{c} \vdash A, A^\perp \quad \vdash B, B^\perp \\ \hline \vdash A \otimes B, A^\perp, B^\perp \quad \vdash C, C^\perp \\ \hline \vdash (A \otimes B) \otimes C, A^\perp, B^\perp, C^\perp \\ \hline \vdash (A \otimes B) \otimes C, A^\perp \wp B^\perp, C^\perp \end{array}$$

are the same, up to the order of the rules. Both are represented by the same proof-net, shown in Fig. 8. The question is to prove the converse of Theorem 2.7, namely that

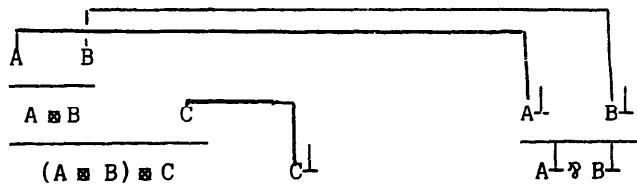


Fig. 8.

a proof-net can always be written as π^- for an appropriate π . The difficulty is that several π 's are possible, and the converse is a very subtle result.

2.9. Theorem. *If β is a proof-net, one can find a proof π in sequent calculus such that $\beta = \pi^-$.*

Proof. By induction on the number of links in β :

(i) If β has exactly one link, then β must be of the form $\overline{A} \quad A^\perp$: in that case, the claim is proved by taking as π the axiom $\vdash A, A^\perp$.

(ii) If β has more than one link, then one of these links must be a *par* or a *times* link (otherwise, β is not connected and it is immediate that, in a nonconnected proof-structure, all trips are short). Hence, there is a terminal formula which is the conclusion of a *par* or a *times* link. In this case, we state the hypothesis that one can find such a terminal formula as the conclusion of a *par* link: write β as

$$\frac{\beta'}{A \quad B} \quad A \otimes B,$$

where β' is the *proof-structure* obtained by restriction. β' has one link less than β . Furthermore, β' is a proof-net: setting all switches for a trip in β' , and setting the additional switch of β on "L", we get a longtrip

$$A^\wedge, \dots, B_v, B^\wedge, \dots, A_v, A \otimes B_v, A \otimes B^\wedge, A^\wedge, \dots$$

in β . But this shows the existence of a longtrip

$$A^\wedge, \dots, B_v, B^\wedge, \dots, A_v, A^\wedge, \dots$$

in β' , and this established the soundness of β' .

The induction hypothesis has built π' such that $\pi'^- = \beta'$. Then, for π one can take the proof:

$$\frac{\pi'}{\vdash A, B, C \quad \otimes}{\vdash A \otimes B, C}$$

(exchanges have not been indicated).

(iii) In this case we assume that there is more than one link, but that no terminal formula is the conclusion of a *par* link. The temptation is to split the proof-net as

follows:

$$\frac{\beta' \quad \beta''}{A \otimes B},$$

where β' and β'' are disjoint subproof-nets of β . Of course, we can try to make such a splitting for each terminal conclusion of a *times* link in β and in simple examples it turns out that if we choose the wrong terminal formula, then the splitting does not work. For instance, the proof-net in Fig. 9 has three terminal \otimes -links but one

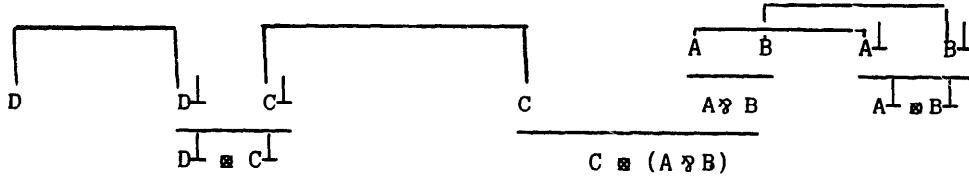


Fig. 9.

of them (the one ending with $A^\perp \otimes B^\perp$) cannot be split. The proof-net can be split at the two other terminal links, and the non-uniqueness of the solution of splitting problems makes it even more complicated. We shall now try to prove the existence of a splitting from which the last case of the theorem will easily follow. Let us remark however that the hypothesis 'no terminal \wp -link' is needed since the net in Fig. 10 cannot be split.



Fig. 10.

2.9.1. Lemma. *Let β be a proof-net and consider (w.r.t. a certain position of the switches) the respective times of passage of the particle through A_v , $A \otimes B^\wedge$, B_v , say t_1, t_2, t_3 , in a given \otimes -link of β ; then $t_2 \in [t_1, t_3]$. In other terms, in case "L", the particle travels as follows:*



and not as:



Proof. Assume, for a contradiction, that $t_3 \in [t_1, t_2]$ and assume that the switch of our \otimes -link is set on, say "L". Then the trip runs as follows:

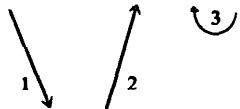
$$A_v, A \otimes B_v, \dots, B_v, A^\wedge, \dots, A \otimes B^\wedge, B^\wedge, \dots, A_v.$$

Now commute the switch to “R” and observe that we get a shorttrip

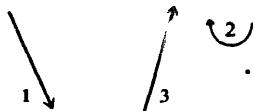
$$A \otimes B_v, B_v, \dots, A \otimes B_v, \dots$$

(The portions in dots in the original trip are common to both trips.) \square

2.9.2. Lemma. Let β be a proof-net and consider (w.r.t. a certain position of the switches) the respective times t_1, t_2, t_3 of passage of the particle through the gates A_v, A^\wedge and B_v of a given \wp -link; then $t_2 \in [t_1, t_3]$. In other terms, in case “L”, the order of passage is



and not:



Proof. Very similar to the proof of Lemma 2.9.1. Assume, for a contradiction, that $t_3 \in [t_1, t_2]$. Then (if our \wp -switch is ‘on “L”’), the trip runs as follows:

$$A_v, A \wp B_v, \dots, B_v, B^\wedge, \dots, A \wp B^\wedge, A^\wedge, \dots, A_v.$$

Setting the switch on the position “R”, we get the shorttrip $A^\wedge, \dots, A_v, A^\wedge$. \square

2.9.3. Definition. Assume that $\frac{A}{A \otimes B}$ is a given \otimes -link in a proof-net β . We define the *empires* eA and eB of A and B as follows: eA consists of those formulas C such that, for any trip leaving A^\wedge at time t_1 and returning at A_v at time t_2 and passing through C^\wedge and C_v at times u_1 and u_2 , we have $u_1, u_2 \in [t_1, t_2]$. The empire eB is defined in a symmetric way.

2.9.4. Facts

- (i) $A \in eA$.
- (ii) $eA \cap eB = \emptyset$.
- (iii) If $C \in eA$, and C is linked to C^\perp by $\overline{C \rightarrow C^\perp}$, then $C^\perp \in eA$.
- (iv) If $C \otimes D \in eA$ is the conclusion of a link $\frac{C}{C \otimes D}$, then $C, D \in eA$.
- (v) If $C \wp D \in eA$ is the conclusion of a link $\frac{C}{C \wp D}$, then $C, D \in eA$.
- (vi) If $\frac{C}{C \otimes D}$ is a \otimes -link distinct from the link $\frac{A}{A \otimes B}$ and if $C \in eA$ (respectively $D \in eA$), then $C \otimes D \in eA$.
- (vii) If $\frac{C}{C \wp D}$ is a \wp -link and $C, D \in eA$, then $C \wp D \in eA$.

Proof. (i): self-evident.

(ii): By Lemma 2.9.1, the intervals A^\wedge, \dots, A_v and B^\wedge, \dots, B_v in a given trip are necessarily disjoint, etc.

(iii): This is because the gates of C^\perp are crossed just one step before or after the gates of C .

(iv): Take a trip with the $C \otimes D$ -switch on "L". Then, since C_v and D^\wedge are passed at one step from $C \otimes D_v$ and $C \otimes D^\wedge$, C_v and D^\wedge belong to the interval A^\wedge, \dots, A_v . Now, imagine that the two consecutive times of passage through D_v and C^\wedge are outside the interval A^\wedge, \dots, A_v : then the interval B^\wedge, \dots, D_v (or $A \otimes B_v, \dots, D_v$) is performed without passing through any of the two links $\frac{A}{A \otimes B} \frac{B}{B \otimes C}$ and $\frac{C}{C \otimes D} \frac{D}{D \otimes A}$. Now, let us commute the switch $\frac{C}{C \otimes D} \frac{D}{D \otimes A}$ to "R". It is immediate (for reasons of symmetry) that D_v belongs to the interval A^\wedge, \dots, A_v of this new trip. However, the part in dots B^\wedge, \dots, D_v (or $A \otimes B_v, \dots, D_v$) is common to both trips, and does not contain A ; a contradiction. We arrive at the conclusion that the four exits of C and D are passed between A^\wedge and A_v .

(v): Assume that the switch of our \wp -link is on "L". Then the times of passage through C_v and C^\wedge being one step from the times of passage through $C \otimes D_v$ and $C \otimes D^\wedge$, it follows that $C_v, C^\wedge \in A^\wedge, \dots, A_v$. Now, assume that the consecutive times of passage through D_v and D^\wedge are outside the interval A^\wedge, \dots, A_v . Then we conclude that the interval B^\wedge, \dots, D_v (or $A \wp B_v, \dots, D_v$) is performed without passing through the links $\frac{A}{A \otimes B} \frac{B}{B \otimes C}$ and $\frac{C}{C \wp D} \frac{D}{D \otimes A}$. Form another trip by setting our \wp -switch to "R" and derive a contradiction as in (iii).

(vi): Consider a trip with the switch $\frac{C}{C \otimes D}$ on "L". Then, by hypothesis, the times of passage through C_v and C^\wedge are within the interval A^\wedge, \dots, A_v . We have the following possibilities, using Lemma 2.9.1:

$$\begin{aligned} & A^\wedge, \dots, C_v, C \otimes D_v, \dots, C \otimes D^\wedge, D^\wedge, \dots, D_v, C^\wedge, \dots, A_v, \\ & A^\wedge, \dots, C \otimes D^\wedge, D^\wedge, \dots, D_v, C^\wedge, \dots, C_v, C \otimes D_v, \dots, A_v, \\ & A^\wedge, \dots, D_v, C^\wedge, \dots, C_v, C \otimes D_v, \dots, C \otimes D^\wedge, D^\wedge, \dots, A_v. \end{aligned}$$

There is no other possibility since $A_\wedge, \dots, D^\wedge, \dots, A^\wedge$ and $A_v, \dots, C \otimes D^\wedge, \dots, A^\wedge$ are easily refuted by commuting our switch to "R". From this we get the result.

(vii): Consider a trip (with our \wp -switch on "L"). Then, by Lemma 2.9.2, the order is as follows:

$$C^\wedge, \dots, D_\wedge, D^\wedge, \dots, C_v, C \wp D_v, \dots, C \wp D^\wedge, C^\wedge$$

and we still have to determine the respective places of A^\wedge and A_v in this pattern. If A^\wedge is between $C \wp D_v$ and $C \wp D^\wedge$, then the only way to ensure that $C \in eA$ is to have

$$A^\wedge, \dots, C \wp D^\wedge, C^\wedge, \dots, C_v, C \wp D_v, \dots, A_v, \dots,$$

i.e., A_v is also in the same interval. If A^\wedge is between C^\wedge and D_v , then the only way to have $C \in eA$ is

$$A^\wedge, \dots, D_v, \dots, C_v, \dots, C^\wedge, \dots, A_v, \dots,$$

i.e., A_v is also in the same interval. Using $D \in eA$ we can conclude in the case where A^\wedge is between D^\wedge and C_v that A_v is also in the same interval. In these three cases, the two exits of $C \wp D$ are between A^\wedge and A_v . \square

2.9.5. Theorem (Trip Theorem). *There is a position of the switches such that eA is exactly the interval A^\wedge, \dots, A_\vee .*

Proof. Each time there is a \wp -link $\frac{C}{C \wp D}$ with exactly one premise in eA , set the corresponding switch to “R” if $C \in eA$ and to “L” if $D \in eA$. The other switches are set arbitrarily. Now we start our trip with A^\wedge , i.e., with an element of eA , and we check that we stay in eA up to the point A_\vee : this can be easily proved by induction since the only way to exit eA would be going from a premise to a conclusion in a \wp -link. In that case, however, only one premise is in eA and the switches have been set such as to move backwards, i.e., to pass from E_\vee to E^\wedge and thus to stay in eA . So the whole portion between A^\wedge and A_\vee is in eA and since it contains eA , we are done. \square

2.9.6. Corollary. *If $C \otimes D$ is ‘above’ A , i.e., is an hereditary premise of A , and if $C \otimes D$ is the conclusion of a \otimes -link $\frac{C}{C \otimes D}$, then $eC \cup eD \subset eA$.*

Proof. Set the switches so that eA is performed between A^\wedge and A_\vee ; since all formulas ‘above’ A belong to eA , we are free to set the switches so that, after A^\wedge , the particle goes from conclusion to premise up to D^\wedge . Then the trip can be visualized as

$$\dots, A^\wedge, \dots, D^\wedge, \dots, D_\vee, \dots, A_\vee, \dots$$

From this, $eD \subset D^\wedge, \dots, D_\vee \subset A^\wedge, \dots, A_\vee = eA$. For similar reasons, $eC \subset eA$. \square

2.9.7. Theorem (Splitting Theorem). *Let β be a proof-net with at least one terminal \otimes -link and no terminal \wp -link. Then it is possible to find a terminal \otimes -link $\frac{A}{A \otimes B}$ such that β is the union of eA , eB , and of the formula $A \otimes B$.*

Proof. Choose a link $\frac{A}{A \otimes B}$ such that $eA \cup eB$ is maximal w.r.t. inclusion. If $A \otimes B$ is not terminal, then below $A \otimes B$ there is a terminal link and, by hypothesis, this link must be of the form $\frac{C}{C \otimes D}$. (Say that, for instance, $A \otimes B$ is above D .) Then, by Corollary 2.9.6, $eA \cup eB \subset eD$, contradicting the assumption of maximality. Now, assume, for a contradiction, that $eA \cup eB \cup \{A \otimes B\} \neq B$. Then we claim the existence of a link $\frac{C}{C \wp D}$ with either $C \in eA$ and $D \in eB$, or $C \in eB$ and $D \in eA$. This link exists because, otherwise, by setting the switches as we did in the proof of the Trip Theorem 2.9.5, we can simultaneously realize $eA = A^\wedge, \dots, A_\vee$ and $eB = B^\wedge, \dots, B_\vee$. In the remaining two steps, the particle is in $A \otimes B$, so $\beta = eA \cup eB \cup \{A \otimes B\}$: a contradiction.

Now the formula $C \wp D$ is above a terminal \otimes -link $\frac{E}{E \otimes F}$ and we may assume that $C \wp D$ is indeed above, say F . Set the switches for a trip such that $F^\wedge, \dots, F_\vee = eF$. In Corollary 2.9.6 it has already been observed that we are free to set the switches as we want above F , and we do it so that, immediately after F^\wedge , the particle runs up to C^\wedge : in particular, our \wp -switch is on “L”. Now the trip looks as follows (using

Lemma 2.9.2)

$$\dots, F^\wedge, \dots, C \wp D^\wedge, C^\wedge, \dots, D_v, D^\wedge, \dots, C_v, C \wp D_v, \dots, F_v, \dots$$

Using the fact that $D \in eB$ and $C \in eA$, the only room for B^\wedge in this picture is between C^\wedge and D_v ; the only possible room for B_v in this picture is between D^\wedge and C_v . But this shows that $eB \subset eF$. By interchanging C and D , we obtain that $eA \subset eF$. But then $eA \cup eB$ cannot be maximal any longer. We have therefore contradicted the assumption $eA \cup eB \cup \{A \otimes B\} \neq \beta$. \square

Proof of Theorem 2.9 (continued). The remaining case is just the case where Theorem 2.9.7 applies, so choose a terminal link $\frac{A}{A \otimes B}$ with the splitting property. We claim that, once the terminal link has been removed, the subsets eA and eB are not linked together. The only possible linkage would be through a \wp -link $\frac{C}{C \wp D}$ with one premise in eA and the other in eB , but then $C \wp D$ would be nowhere. So it makes sense to speak of the respective restrictions β' and β'' of β to eA and eB respectively. The fact that β' and β'' are in turn proof-nets is a trifle compared with the complexity of the proof of the Splitting Theorem and is therefore omitted! If we associate, using the induction hypothesis, with β' and β'' proofs π' and π'' in linear sequent calculus of $\vdash A, C$ and $\vdash B, D$ respectively such that $\pi'^- = \beta'$ and $\pi''^- = \beta''$, then we can define π as

$$\frac{\pi' \quad \pi''}{\vdash A \otimes B, C, D} \otimes$$

and, clearly, $\pi^- = \beta$. \square

2.2. The cut-rule

The cut-rule has been omitted because, geometrically speaking, it is very close to the \otimes -rule. The cut-rule can be written as a link

$$\underline{A} \quad \underline{A^\perp}$$

with two premises and no conclusion. Trips through this link proceed in the obvious way: after A_v go to $A^{\perp\wedge}$, after A_v^\perp go to A^\wedge . Now this is (up to inessential details) as if one were working with a link

$$\frac{A \quad A^\perp}{A \otimes A^\perp}$$

with $A \otimes A^\perp$ terminal (the position of the corresponding switch does not matter). We decide to write the cut-rule as

$$\frac{A \quad A^\perp}{\text{CUT}},$$

where CUT is a symbol (not a formula) which is used so that the cut-rule looks formally like a \otimes -link, the only difference being that since CUT is not a formula, CUT is necessarily terminal. Then this link is treated exactly as a \otimes -link and our main theorem immediately extends to the calculus with cv' . In particular, the phenomenon of splitting (Theorem 2.9.7) involves either a \otimes -link or a CUT-link.

2.3. Cut-elimination

Take β , a proof-net in the multiplicative fragment with cut and select a particular CUT-link $\frac{A \quad A^\perp}{\text{CUT}}$. We define a *contractum* β' of β (w.r.t. this particular link) as follows:

- (i) If A and A^\perp are the respective conclusions of dual multiplicative links, i.e., if β ends with

$$\frac{\begin{array}{c} \vdots \quad \vdots \\ B \quad C \end{array} \quad \begin{array}{c} \vdots \quad \vdots \\ B^\perp \quad C^\perp \end{array}}{\begin{array}{c} B m C \\ B^\perp m^\perp C^\perp \end{array}} \quad \text{CUT}$$

with m, m^\perp dual multiplicatives, then replace this part of β by

$$\frac{\begin{array}{c} \vdots \quad \vdots \\ B \quad B^\perp \end{array} \quad \begin{array}{c} \vdots \quad \vdots \\ C \quad C^\perp \end{array}}{\begin{array}{c} \text{CUT} \\ \text{CUT} \end{array}}.$$

- (ii) If A is the conclusion of an axiom-link, i.e., β ends with

$$\frac{\begin{array}{c} \vdots \\ A^\perp \end{array} \quad \begin{array}{c} \vdots \\ A \end{array} \quad \begin{array}{c} \vdots \\ A^\perp \end{array}}{\begin{array}{c} \text{CUT} \end{array}}$$

unify the two occurrences of A^\perp so that one gets

$$\begin{array}{c} \vdots \\ A^\perp. \\ \vdots \end{array}$$

- (iii) If A^\perp is the conclusion of an axiom-link, we define β' symmetrically. When both A and A^\perp are conclusions of axiom-links, i.e., in the situation

$$\frac{\begin{array}{c} \vdots \\ A^\perp \end{array} \quad \begin{array}{c} \vdots \\ A \end{array} \quad \begin{array}{c} \vdots \\ A^\perp \end{array} \quad \begin{array}{c} \vdots \\ A \end{array}}{\begin{array}{c} \text{CUT} \end{array}},$$

then we have a conflict between (ii) and (iii). However, both cases lead to

$$\begin{array}{c} \vdots \\ A^\perp \quad A \\ \vdots \quad \vdots \end{array}$$

2.10. Proposition. (1) *If β is a proof-net and β' is a contractum of β , then β' is a proof-net.*

(2) *β' is strictly smaller than β , i.e., has strictly less formulas.*

Proof. We prove (2) first, then (1).

(2): The size $s(\beta)$ of a proof-net is the number of formulas of β including the symbol CUT. The inclusion of these cut-symbols is the most elegant solution. In case (i) the size is diminished by 1; in cases (ii) and (iii) it is diminished by three units. There is however a hidden trap in this apparently trivial proof: when A is the conclusion of an axiom-link, who tells us that the two occurrences of A^\perp are distinct ? In fact, they can be the same if β is the proof-structure

$$\frac{\overline{A \quad A^\perp}}{\text{CUT}}$$

in which case β' is β . But this particular proof-structure is obviously not a proof-net since it has a shorttrip. This shows the extreme importance of being a proof-net.

(1): Assume that we are in case (i) (cases (ii) and (iii) are immediate) and that the multiplicative m is \otimes . Set the switches of β' for a trip. This induces a switching of β (our two multiplicative links being switched on “L”). Then the trip works as follows (cf. Lemmas 2.9.1 and 2.9.2):

$$\begin{aligned} & B_v, B \otimes C_v, B^\perp \wp C^{\perp\wedge}, B^{\perp\wedge}, \dots, C_v^\perp, C^{\perp\wedge}, \dots, B_v^\perp, B^\perp \wp C_v^\perp, \\ & B \otimes C^\wedge, C^\wedge, \dots, C_v, B^\wedge, \dots, B_v. \end{aligned}$$

Hence, in β' , the particle travels as follows:

$$B_v, B^{\perp\wedge}, \dots, C_v^\perp, C^\wedge, \dots, C_v, C^{\perp\wedge}, \dots, B_v^\perp, B^\wedge, \dots, B_v$$

which is a longtrip. \square

2.11. Proposition. *Say that β reduces to β' (notation: $\beta \text{ red } \beta'$) when β' is a hereditary contractum of β . Then, reduction satisfies the Church-Rosser property.*

Proof. Essentially, this proposition says that if we contract two distinct cuts of β , then the result does not depend on the order of application of the contractions. This is practically immediate. \square

2.12. Theorem (Strong Normalization). *A proof-net of size n normalizes into a cut-free proof-net in less than n steps; the result, which does not depend on the order of application of the contractions, is called the normal form of our proof-net.*

2.13. Remark. The ‘ n steps’ of the theorem suggest a sequential procedure for the normalization process. It is, of course, more natural to think of a parallel procedure,

namely to work independently on each CUT. One of the astounding properties of the multiplicative fragment is that everything works perfectly well, i.e., that there is not the slightest problem of synchronization.

The problem is now to extend the theory of proof-nets to the full language. At the present moment (September 1986) there is no extant extension of the same quality as what exists for the multiplicative fragment. Moreover, we doubt that anything really convincing might ever be done for the additives. For the rest of the language, what we present below can surely be improved....

The general pattern we shall now introduce is that of a *proof-box*. Roughly speaking, proof-boxes are synchronization marks in the proof-net. They can also be seen as moments where we restore the sequent (i.e., the sequential!) structure. Their use is therefore a bridle to parallelism and, for that reason, one must try to limit their use. In particular, reasonable improvements of the concept of proof-net showing that some rules can be written without boxes will be of great practical interest (however, we believe that $(\&)$ cannot be written without boxes).

How to *make* a box depends on the particular syntax, the particular rules we want to express, and is of no interest right now. What concerns us now is how to *use* a box. When made, a box looks as shown in Fig. 11, i.e., a black ‘thing’ with

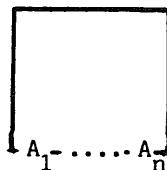


Fig. 11.

n outputs/inputs, $n \neq 0$. What is inside the square is the building process of the box, which is irrelevant: boxes are treated in a perfectly modular way: we can use the box \mathbb{B} in β without knowing its contents, i.e., another box \mathbb{B}' with exactly the same n doors A_1, \dots, A_n would do as well. This is the principle of the *black box*: in order to check the correctness of a proof-net involving a box \mathbb{B} , we have to check the proof-net without knowing anything about \mathbb{B} and check \mathbb{B} itself. Now, what is the criterion for the correctness of a proof-net involving a box \mathbb{B} ? Simply, the box is treated as any potential proof-net with conclusions A_1, \dots, A_n .

2.14. Definition. The *multiplicative fragment with proof-boxes* admits, besides the links already acknowledged, a new kind of link shown in Fig. 12, where A_1, \dots, A_n are arbitrary formulas and $n \neq 0$. With such a box is associated a switch: a position of the switch consists in a permutation of n which is cyclic, i.e., $1, \sigma(1), \sigma^2(1), \dots, \sigma^{n-1}(1)$ are all distinct. The trip is executed as follows: after A_i^\wedge go to $A_{\sigma(i)}^\vee$.

2.15. Theorem. *Theorems 2.7 and 2.9 can be extended to the calculus with boxes; the*

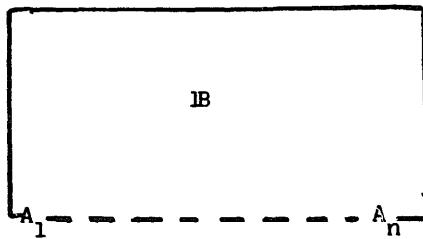


Fig. 12.

use of a proof-box with doors A_1, \dots, A_n in a proof-net corresponds to the use of the axiom $\vdash A_1, \dots, A_n$.

Proof. The extension of the results to this new context offers no difficulty. In fact, proof-boxes have more or less the same structure as the logical axiom $A \quad A^\perp$. For instance, let us look at the analogue of Fact 2.9.4(iii): if one door A_k of B belongs to eA , then any other door A_i of B does so too. (*Proof:* First observe that a trip is performed by putting together slices $A_{iv}, \dots, A_i^\wedge$ because if outside the box one could have something like $A_{iv}, \dots, A_j^\wedge$ with $j \neq i$, then, by appropriately choosing $\sigma(\sigma(j)=i)$, one would get the shorttrip $A_{iv}, \dots, A_j^\wedge, A_{iv}$. Now, if A^\wedge and A_v are located in $A_{iv}, \dots, A_i^\wedge$ and $A_{jv}, \dots, A_j^\wedge$ respectively, we claim that $i=j$; suppose otherwise, then take $\sigma(i)=j$: the two doors of A_k are not in $A_{iv}, \dots, A_j^\wedge$. Finally, the only possible location of A^\wedge and A_v within $A_{iv}, \dots, A_i^\wedge$ is $A_{iv}, \dots, A_v, \dots, A^\wedge, \dots, A_i^\wedge$ and from this, we easily conclude that all the doors of B are in eA .) Apart from this analogue, there is very little novelty in this generalization. \square

2.4. The system PN1

The system PN1 is a proof-net system for linear propositional calculus based on the idea of putting all contextual rules (i.e., rules which in sequent calculus depend on the context in sequent calculus) into proof-boxes without trying to limit the number of boxes. Besides boxes and the multiplicative links, there will be a certain number of links corresponding to unary rules

$$\frac{A}{B},$$

such links are so harmless (trip: from A_v go to B_v , from B^\wedge go to A^\wedge) that they practically do not alter the multiplicative paradise.

Axioms:

$$\overline{A} \quad A^\perp \quad \text{logical axiom,}$$

$$1 \quad \text{axiom for } 1,$$

$$\boxed{T - C} \quad \text{axiom for } T.$$

Technically speaking, these three axioms are boxes, in which we distinguish *front doors* (A and A^\perp in the logical axiom, 1 in the axiom for 1 , \top in the last case), and *auxiliary doors* (C in the last case).

Unary links:

$$\frac{A}{A \oplus B} 1\oplus, \quad \frac{B}{A \oplus B} 2\oplus, \quad \frac{A}{?A} D?.$$

Binary links:

$$\frac{A \quad B}{A \otimes B} \otimes, \quad \frac{A \quad B}{A \wp B} \wp,$$

$$\frac{?A \quad ?A}{?A} C?, \quad \frac{A \quad A^\perp}{CUT}.$$

Box formation: From proof-nets β' and β'' whose respective conclusions (CUT is not counted) are A, C and B, C , we form a box whose conclusions are $A \& B, C$ (see Fig. 13: front door: $A \& B$; auxiliary door: C).

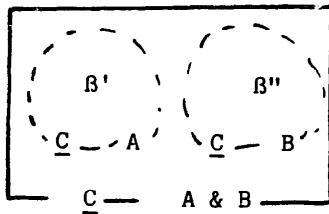


Fig. 13.

From a proof-net β with conclusions C (CUT is not counted), we form a box with conclusion C, \perp (respectively $C, ?A$) (see Fig. 14: front door: \perp (respectively $?A$); auxiliary doors: C).

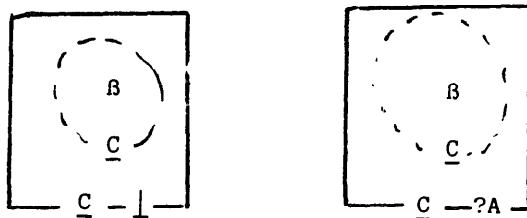


Fig. 14.

From a proof-net β whose conclusions (CUT is not counted) are $A, ?B$, we form a box whose conclusions are $!A, ?B$ (see Fig. 15: front door: $!A$, auxiliary doors: $?B$).

We recall that, geometrically speaking, the rule (CUT) behaves like $(\otimes)^4$ and that rule $(C?)$ behaves like (\wp) . It is not difficult to translate propositional linear

⁴ It is convenient to consider the CUT link symmetric, i.e., there is no 'left' or 'right' in the rule as in the logical axiom.

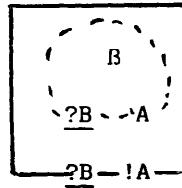


Fig. 15.

logic within this system, the only remark to make being that the rule ($W?$) is now done with the logical axioms. The fact that this translation is faithful offers no difficulty.

2.5. The system PN2

We finally direct our attention to the predicate case. Here, however, we make a different syntactic choice, moving from first-order to second-order. It was natural to deal with first-order linear logic to get a completeness theorem as obtained in Section 1. Already at that moment we did not spend too much time with the quantifiers! As the paper continues it focuses more and more on computational aspects, so it is more and more interesting to consider second-order logic in the spirit of the system F of [1]. So we shall develop here the system PN2 of second-order propositional logic. First-order quantifiers will from now on disappear from our world. The reader who would like to consider second-order predicate calculus would not encounter the slightest problem, just as the results on F were immediately transferable to Takeuti's system. Second-order propositional logic (linear version) is obtained by considering propositional variables a, b, c, \dots (instead of arbitrary constants) together with the quantifiers $\wedge a.$ and $\vee a..$ The crucial concept is that of substitution of a proposition B for a propositional variable a in A , denoted $A[B/a]$: Replace all occurrences of a by B , all occurrences of a^\perp by B^\perp . In terms of sequent calculus, the rules for second-order quantification can be written as

$$\frac{\vdash A, B}{\vdash \wedge a.A, b} \wedge, \quad \frac{\vdash A[B/a], C}{\vdash \vee a.A, C} \vee.$$

In (\wedge) , there is the familiar restriction: a not free in B .

The rules for quantifiers are handled as follows:

- on the one hand, the unary link

$$\frac{A[B/a]}{\vee a.A} \vee;$$

- on the other hand, the box-formation scheme: from a proof-net β whose conclusions B, A are such that a does not occur free in B , we form a box whose outputs are B and $\wedge a.A$ (see Fig. 16).

2.16. Definition (terminology). PN1 and PN2 have already been introduced; PN0 is the subsystem of PN1 concerned with the multiplicative fragment only including

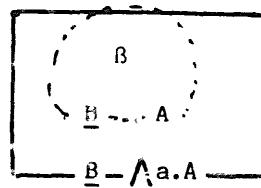


Fig. 16.

cut. $\text{PN}0^-$ is the cut-free part of $\text{PN}0$; similarly, $\text{PN}1^-$ and $\text{PN}2^-$ are the cut-free parts of $\text{PN}1$ and $\text{PN}2$.

2.6. Discussion

Here we shall shortly discuss the possibility of improving the syntax by removing or transforming some boxes.

(i) The box for $!$ is an absolute one; later on, we shall write no contraction rules making this scheme commute with others. Hence, the box, viewed as an interruption of the linear features, should not be touched. Even if, by some sort of miracle, it were possible to erase such boxes, this would mean that one can reconstruct them in a unique way, so why not directly note them?

(ii) The boxes for weakening seem of very limited interest. The problem is that if we admit configurations like

$$\begin{array}{c} \beta \\ | \\ A \end{array} \quad \perp,$$

then we are forced to accept certain shorttrips from which endless complications start. However, the only kind of shorttrip that is definitely bad is the one including only one gate for a formula, e.g. A^\wedge but not A_\vee , and the inclusion of unboxed weakening would not introduce these bad boxes. A nice criterion for soundness for this variant is however not yet known.

(iii) The box for \wedge seems removable too; the same comments hold as in the case of the weakening boxes.

(iv) Finally, the case of $\&$ -boxes is the most delicate; when we contract such boxes, there is a phenomenon of duplication, and removing the boxes would mean that we have a way to write the two boxes in Fig. 17 as the same proof-net. This goal seems out of reach. However, see Section 6 for what could be a solution by means of families of proof-structures.

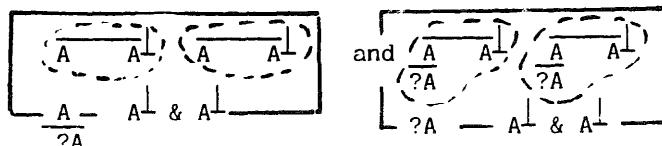


Fig. 17.

3. The coherent semantics

In Section 1, we developed the semantics of linear logic in the Tarskian style, i.e., by explaining the formulas. But there is another semantic tradition, amounting by Heyting's semantics of proofs, which consists in modelling the proofs themselves. In this tradition, proofs are considered as functions, relations, etc. on some kind of constructive space, which has often been viewed as Scott domains. In previous work, we have simplified Scott domains in order to get what we called *qualitative domains*. The *binary* ones, rebaptized *coherent spaces*, will form the core of our semantics of proofs.

3.1. Definition. A *coherent space* X is a set satisfying the following conditions:

- (i) $a \in X \wedge b \subset a \Rightarrow b \in X$.
- (ii) Say that $a, b \in X$ are *compatible* w.r.t. X when $a \cup b \in X$; then, if A is any subset of X formed of pairwise compatible elements, we have $\bigcup A \in X$. In particular, $\emptyset \in X$.

3.2. Definition. Assume that X is a coherent space. The *web* of X , $W(X)$ is a reflexive, unoriented graph, defined by

- (i) the domain $|X|$ of the graph is $\bigcup X = \{z ; \{z\} \in X\}$;
- (ii) the linkage relation (*coherence modulo* X) is defined as follows:

$$x \square y \text{ [mod } X] \text{ iff } \{x, y\} \in X.$$

3.3. Proposition. The map associating to any coherent space X its web $W(X)$ is a bijection between coherent spaces and reflexive unoriented graphs. The graph may be recovered from its web by means of the formula

$$a \in X \Leftrightarrow a \subset |X| \wedge \forall x, y \in a : x \square y \text{ [mod } X].$$

Proof. More or less immediate. \square

3.4. Remark. In general, the webs stay denumerable and effective, whereas the spaces have the power of the continuum. So, most of the time we deal with webs. Instead of coherence, it is sometimes more convenient to consider *strict coherence*:

$$x \frown y \text{ [mod } X] \text{ iff } x \square y \text{ [mod } X] \text{ and } x \neq y.$$

The following relations are also of interest:

- *strict incoherence*: $x \curlyeq y \text{ [mod } X]$ iff $\neg(x \square y \text{ [mod } X])$;
- *incoherence*: $x \bowtie y \text{ [mod } X]$ iff $\neg(x \frown y \text{ [mod } X])$.

We shall now proceed with the definition of the interpretation of the operations of linear logic in terms of coherent spaces. A large attention will be paid to *canonical isomorphisms* (indicated by \sim). It is however necessary to attract the reader's attention to the two following points:

(i) \sim is not a connective of linear logic; in particular, it is strictly stronger than \multimap . In fact, it would be easy to define a coherent semantics for something like \sim , but we would get problems in other stages, i.e., the phase semantics and the proof-nets.

(ii) Some canonical isomorphisms are omitted: this is because coherent spaces do not catch all of linear logic: typically, 1 and \perp are the same coherent space for stupid reasons, but their equivalence (translated as $1 \multimap \perp$) is not provable. So, writing the isomorphisms would have been misleading.

3.5. Definition (linear negation). If X is a coherent space, its linear negation X^\perp is defined by

$$|X^\perp| = |X|,$$

$$x \odot y \text{ [mod } X^\perp] \text{ iff } x \asymp y \text{ [mod } X].$$

In other terms, the operation *nil* exchanges coherence and incoherence. We clearly have $X^{\perp\perp} = X$, i.e., *nil* is involutive. The existence of a constructive involution is a tremendous improvement on intuitionistic logic. As we shall proceed with the other connectives, it will be our first task to check the De Morgan equalities, which will justify, at the level of the coherent semantics, the restriction of *nil* to atoms.

3.6. Definition (multiplicatives). The name itself comes from the coherent semantics: the three multiplicatives are variations on the theme of the cartesian product:

$$|X \otimes Y| = |X \wp Y| = |X \multimap Y| = |X| \times |Y|,$$

$$(x, y) \odot (x', y') \text{ [mod } X \otimes Y] \text{ iff } x \odot x' \text{ [mod } X] \text{ and } y \odot y' \text{ [mod } Y],$$

$$(x, y) \cap (x', y') \text{ [mod } X \wp Y] \text{ iff } x \cap x' \text{ [mod } X] \text{ or } y \cap y' \text{ [mod } Y],$$

$$(x, y) \odot (x', y') \text{ [mod } X \multimap Y] \text{ iff } (x \cap x' \text{ [mod } X] \Rightarrow y \cap y' \text{ [mod } Y]) \text{ and}$$

$$(x \odot x' \text{ [mod } X] \Rightarrow y \odot y' \text{ [mod } Y]).$$

Observe that \otimes is defined in terms of \odot , whereas \wp is defined in terms of \cap , and \multimap is defined in terms of preservation properties, which is in the spirit of an implication, linear or not.

The De Morgan equalities are easily verified:

$$(X \otimes Y)^\perp = X^\perp \wp Y^\perp, \quad (X \wp Y)^\perp = X^\perp \otimes Y^\perp, \quad X \multimap Y = X^\perp \wp Y \text{ etc.}$$

The *commutativity* of the multiplicatives is expressed by

$$X \otimes Y \sim Y \otimes X, \quad X \wp Y \sim Y \wp X, \quad X \multimap Y \sim Y^\perp \multimap X^\perp.$$

The *associativity* of the multiplicatives is expressed by

$$X \otimes (Y \otimes Z) \sim (X \otimes Y) \otimes Z, \quad X \wp (Y \wp Z) \sim (X \wp Y) \wp Z,$$

$$X \multimap (Y \multimap Z) \sim (X \otimes Y) \multimap Z, \quad X \multimap (Y \wp Z) \sim (X \multimap Y) \wp Z.$$

3.7. Definition (the unit coherent space). Up to isomorphism, there is exactly one coherent space whose web consists of one point, say 1 . In particular, this space is equal to its linear negation. The space will be indifferently denoted 1 or \perp since it will stand as the interpretation of these constants. We write the isomorphisms expressing the *neutrality* of the unit space, choosing between the two notations 1 and \perp the one which has also a logical meaning:

$$X \otimes 1 \sim X, \quad X \wp \perp \sim X, \quad 1 \multimap X \sim X, \quad X \multimap \perp \sim X^\perp.$$

3.8. Definition (the additives). Here too the name comes from the coherent semantics: the two additives are variations on the theme of the direct sum:

$$|X \& Y| = |X \oplus Y| = |X| + |Y| = \{0\} \times |X| \cup \{1\} \times |Y|,$$

$$(0, x) \subset (0, x') [\text{mod } X \& Y] \text{ or } [\text{mod } X \oplus Y] \text{ iff } x \subset x' [\text{mod } X],$$

$$(1, y) \subset (1, y') [\text{mod } X \& Y] \text{ or } [\text{mod } X \oplus Y] \text{ iff } y \subset y' [\text{mod } Y],$$

$$(0, x) \cap (1, y) [\text{mod } X \& Y] \text{ for all } x \in |X| \text{ and } y \in |Y|,$$

$$(0, x) \cup (1, y) [\text{mod } X \oplus Y] \text{ for all } x \in |X| \text{ and } y \in |Y|.$$

X and Y have been recopied as if they were in the direct sum; now there are only two good-taste solutions as to coherence between X and Y : either always “yes” ($\&$) or always “no” (\oplus).

The De Morgan equalities are immediate:

$$(X \& Y)^\perp = X^\perp \oplus Y^\perp, \quad (X \oplus Y)^\perp = X^\perp \& Y^\perp.$$

The *commutativity* of the additives is expressed by

$$X \& Y \sim Y \& X, \quad X \oplus Y \sim Y \oplus X.$$

The *associativity* of the additives is expressed by

$$X \& (Y \& Z) \sim (X \& Y) \& Z, \quad X \oplus (Y \oplus Z) \sim (X \oplus Y) \oplus Z.$$

The *distributivity* of the multiplicatives w.r.t. the additives is expressed by

$$X \otimes (Y \oplus Z) \sim (X \otimes Y) \oplus (X \otimes Z),$$

$$X \wp (Y \& Z) \sim (X \& Y) \wp (X \& Z),$$

$$X \multimap (Y \& Z) \sim (X \multimap Y) \& (X \multimap Z),$$

$$(X \oplus Y) \multimap Z \sim (X \multimap Y) \& (X \multimap Z).$$

The *semi-distributivity* of the multiplicatives w.r.t. the additives is the fact that

between $X \otimes (Y \& Z)$ and $(X \otimes Y) \& (X \otimes Z)$,

between $(X \wp Y) \oplus (X \wp Z)$ and $X \wp (Y \oplus Z)$,

between $(X \multimap Y) \oplus (X \multimap Z)$ and $X \multimap (Y \oplus Z)$,

between $(X \multimap Z) \oplus (Y \multimap Z)$ and $(X \& Y) \multimap Z$

it is possible to construct a bijective mapping from the web of the left-hand space to the web of the right-hand space, preserving coherence. In general, these maps are not isomorphisms.

3.9. Definition (the null coherent space). This space has a void web. We shall denote it by $\mathbf{0}$ or \top , with the same hidden intentions as in the case of the unit coherent space: the *neutrality* of the void space w.r.t. additives is expressed by

$$X \oplus \mathbf{0} \sim X, \quad X \& \top \sim X.$$

The fact that they *absorb* multiplicatives is expressed by

$$X \otimes \mathbf{0} \sim \mathbf{0}, \quad X \wp \top \sim \top, \quad \mathbf{0} \multimap X \sim \top, \quad X \multimap \top \sim \top.$$

3.10. Definition (the exponentials).

$$|!X| = \{a ; a \in X \text{ and } a \text{ finite}\}, \quad a \subset b \text{ [mod } !X] \text{ iff } a \cup b \in X,$$

$$|?X| = \{a ; a \in X^\perp \text{ and } a \text{ finite}\}, \quad a \cap b \text{ [mod } ?X] \text{ iff } a \cup b \notin X^\perp.$$

The De Morgan formulas are immediate:

$$(!X)^\perp = ?(X^\perp), \quad (?X)^\perp = !(X^\perp).$$

The *distributivity* isomorphisms are

$$!(X \& Y) \sim (!X) \otimes (!Y), \quad ?(X \oplus Y) \sim (?X) \wp (?Y).$$

The following can be taken as definitions of $\mathbf{1}$ and \perp :

$$\mathbf{1} \sim \mathbf{0}, \quad \perp \sim \top.$$

We now have to deal with quantifiers. First-order quantifiers have a boring coherent semantics; hence, we concentrate on second-order propositional quantifiers, which will be handled by means of the category-theoretic methods introduced in [5] which we quickly recall in the following items (i)-(v):

(i) Coherent spaces form a category COH by taking as morphisms from X to Y the set $\text{COH}(X, Y)$ of all injective functions from $|X|$ to $|Y|$ such that

$$\forall x, y \in |X| \quad x \subset y \text{ [mod } X] \leftrightarrow f(x) \subset f(y) \text{ [mod } Y].$$

Associated with such a morphism are two (linear) maps:

- f^+ from X to Y : $f^+(a) = \{f(z) ; z \in a\}$,
- f^- from Y to X : $f^-(b) = \{z ; f(z) \in b\}$.

(ii) Now, if $A[\alpha_1, \dots, \alpha_n]$ is a formula of second-order propositional calculus, we can define an associated functor (denoted $A[X_1, \dots, X_n]$, $A[f_1, \dots, f_n]$) from COH^n to COH :

(1) when A is one of the constants $\mathbf{0}$, $\mathbf{1}$, \top , or \perp , then $A[X]$ is the coherent space already described in Definitions 3.7 or 3.9, whereas $A[f]$ is the identity morphism of this space;

- (2) when A is the propositional variable α_i , then $A[X] = X_i$ and $A[f] = f_i$;
- (3) when A is α_i^+ , then $A[X] = X_i^+$ and $A[f] = f_i$;
- (4) when A is $B \otimes C$, then $A[X] = B[X] \otimes C[X]$ and $A[f](x, y) = (B[f](x), C[f](y))$;
- (5) when A is $B \wp C$, then $A[X] = B[X] \wp C[X]$ and $A[f](x, y) = (B[f](x), C[f](y))$;
- (6) when A is $B \& C$, then $A[X] = B[X] \& C[X]$ and

$$A[f](0, x) = (0, B[f](x)), \quad A[f](1, y) = (1, C[f](y));$$

- (7) when A is $B \oplus C$, then $A[X] = B[X] \oplus C[X]$ and

$$A[f](0, x) = (0, B[f](x)), \quad A[f](1, y) = (1, C[f](y));$$

- (8) when A is $!B$, then $A[\lambda] = !B[X]$ and $A[f](a) = B[f]^+(a)$;
- (9) when A is $?B$, then $A[X] = ?B[X]$ and $A[f](a) = B[f]^+(a)$.

The definition of $A[X]$ when A is of the form $\wedge \beta.B$ or $\vee \beta.B$ will be given soon. For the definition of $A[f]$ in both cases, we have to define $A[f](X_0, z) = (X'_0, z')$, where X'_0 and z' are obtained as follows: Let $y = B[g, f](z)$ where g is the identity of X_0 ; then, w.r.t. the functor $A[., Y]$ ($f \in \text{COH}(X, Y)$), y has a normal form $y = A[h, \text{id}_Y](z')$ for a suitable X'_0 , a suitable $h \in \text{COH}(X_0, X'_0)$ and a suitable z' such that (X'_0, z') is in the trace of $A[., Y]$.

- (iii) The functors defined in that way preserve
 - direct limits,
 - pull-backs.

Unfortunately, preservation of kernels is lost when dealing with exponentials and this is one of the reasons for looking for variants of the exponentials.

(iv) For such functors, we have a *Normal Form Theorem* [5, Theorem 1.3]: let F be a functor from COH to COH preserving direct limits and pull-backs; if $z \in |F(X)|$, then it is possible to find a finite coherent space X_0 together with an $f \in \text{COH}(X_0, X)$ and a $z_0 \in |F(X_0)|$ such that

- (1) $z \in F(f)(z_0)$;
- (2) if $Y, g \in \text{COH}(Y, X)$ and $z_1 \in |F(Y)|$ are such that $z = F(g)(z_1)$, then there is a unique $h \in \text{COH}(X_0, Y)$ such that $z = F(g)(z_1)$ and $f = gh$.

(v) A *trace* of F is a set T such that

- (1) the elements of T are pairs (X, z) , with X finite coherent space and $z \in |F(X)|$;
- (2) given any coherent space Y and any $y \in |F(Y)|$, there is a *unique* $(X, z) \in T$ and a morphism $f \in \text{COH}(X, Y)$ such that $y = F(f)(z)$. One uses the notation $\text{Tr}(F)$ to denote an arbitrary trace of F , chosen once and for all; the actual choice of the trace does not matter.

3.11. Definition (the quantifiers). Assume that F is a functor from COH to COH preserving direct limits and pull-backs; then $\wedge F$ and $\vee F$ are defined as follows:

- (1) $|\wedge F|$ consists of those (X, z) in $\text{Tr}(F)$ such that, for all f_1, f_2 in $\text{COH}(X, Y)$: $F(f_1)(z) \cap F(f_2)(z) \neq \emptyset$ [mod $F(Y)$]. Further, $(X, z) \subset (X', z')$ [mod $\wedge F$]

iff, for all Y , for all $f \in \text{COH}(X, Y)$ and $f' \in \text{COH}(X', Y)$: $F(f)(z) \square F(f')(z') \pmod{F(Y)}$.

(2) $|\vee F|$ consists of those (X, z) in $\text{Tr}(F)$ such that, for all Y and all f_1, f_2 in $\text{COH}(X, Y)$: $F(f_1)(z) \asymp F(f_2)(z) \pmod{F(Y)}$. Further, $(X, z) \square (X', z') \pmod{\vee F}$ iff for some Y , for some $f \in \text{COH}(X, Y)$ and $f' \in \text{COH}(X', Y)$: $F(f)(z) \square F(f')(z) \pmod{F(Y)}$.

3.12. Remarks. (i) $\wedge \beta.A[\beta, X]$ and $\vee \beta.A[\beta, X]$ are defined as respectively, $\wedge F$ and $\vee F$, where F is the functor $A[., X]$.

(ii) The quantifiers involved in Definition 3.11 can be bounded. This was indeed shown in [5], and *binary* qualitative domains (now called coherent spaces) were introduced just to ensure that.

(iii) Computations made in the same paper show that

$$\wedge \alpha. \alpha = \mathbf{0}, \quad \wedge \alpha. \alpha \multimap \alpha \sim \mathbf{1}.$$

(In fact, it is shown that $\wedge \alpha. \alpha \Rightarrow \alpha \sim \mathbf{1}$, from which we easily get $\wedge \alpha. \alpha \multimap \alpha \sim \mathbf{1}$.)

(iv) The interpretation of existence is a pure De-Morganization; in particular, $|\vee F|$ has an unexpected definition which has deep consequences, and is presumably far from being well understood.

3.13. Example (some isomorphisms). Among the many isomorphisms involving quantifiers, let us mention:

(1) the De Morgan equalities:

$$(\wedge \alpha. A)^\perp = \vee \alpha. A^\perp, \quad (\vee \alpha. A)^\perp = \wedge \alpha. A^\perp;$$

(2) the *distributivity* formulas:

$$\wedge \alpha. (A[\alpha] \wp B) \sim (\wedge \alpha. A[\alpha]) \wp B, \quad \vee \alpha. (A[\alpha] \otimes B) \sim (\vee \alpha. A[\alpha]) \otimes B,$$

$$\wedge \alpha. (A[\alpha] \multimap B) \sim (\vee \alpha. A[\alpha]) \multimap B, \quad \wedge \alpha. (A \multimap B[\alpha]) \sim A \multimap \wedge \alpha. B[\alpha];$$

(3) the *associativity* formulas:

$$\wedge \alpha. (A[\alpha] \& B) \sim (\wedge \alpha. A) \& B, \quad \vee \alpha. (A[\alpha] \oplus B) \sim (\vee \alpha. A[\alpha]) \oplus B;$$

etc. To remember them, recall that \wedge and \vee are to some extent generalized additives, and that \wedge looks like $\&$, \vee looks like \oplus .

Our next goal is to define the semantics of the proofs of second-order linear propositional logic. For this, it is simpler to stay with sequents in order to avoid combining the novelty of the semantics with the novelty of the syntax!

3.14. Remark (general pattern). Our goal is to interpret a proof π of a sequent $\vdash A$ in linear sequent calculus. For this, we shall make a certain number of conventions:

(1) We consider A to be made out of *closed* propositions; if not, substitute for the free propositions of A arbitrary coherent spaces. This means that we are in fact

working with additional constants, namely X for each coherent space X ; but this is enough for these trifles.

(2) We shall identify a formula with its associated coherent space; a convention that saves boring extra symbols like $*$, etc.

(3) However, we use $*$ for the interpretation of π ; hence, π^* is an object of the coherent space that would interpret the *par* of the sequence A . In other terms, π^* will be made of sequences $z \in A$. By this we mean, in case A is A_1, \dots, A_n that z is of the form z_1, \dots, z_n , and $z_1 \in A_1, \dots, z_n \in A_n$. Strict coherence modulo A is defined by

$$z \sim z' \text{ [mod } A] \text{ iff } z_i \sim z'_i \text{ [mod } A_i] \text{ for some } i.$$

Of course, one has to check that the subsets π^* are made of pairwise coherent elements. (For readability, we add a symbol \vdash in front of the sequences z .)

3.15. Definition. π^* is defined by induction on the proof π of $\vdash A$. We simultaneously check the coherence of π^* modulo A .

(i) If π is the axiom $\vdash B, B^\perp$, then $\pi^* = \{\vdash z, z ; z \in |B|\}$. Observe that when $z \neq z'$, either $z \sim z' \text{ [mod } B]$ or $z \sim z' \text{ [mod } B^\perp]$.

(ii) If π is obtained from π' and π'' by the cut-rule

$$\frac{\vdash B, C \quad \vdash B^\perp, D}{\vdash C, D},$$

then $\pi^* = \{\vdash z', z'' ; \exists y \in |B| (\vdash y, z' \in \pi'^* \text{ and } \vdash y, z'' \in \pi''^*)\}$. To show the compatibility of two different points $\vdash z', z''$ and $\vdash t', t''$ of π^* , assume that y, x are such that $\vdash y, z'$ and $\vdash x, t' \in \pi'^*$, and $\vdash y, z''$ and $\vdash x, t'' \in \pi''^*$. Then, either $y \asymp x \text{ [mod } B]$ or $y \asymp x \text{ [mod } B^\perp]$. In the first case, $\vdash z' \sim z'' \text{ [mod } C]$; in the other one, $\vdash t' \sim t'' \text{ [mod } D]$; and in both cases, $\vdash z', t' \sim z'', t'' \text{ [mod } C, D]$.

The most important semantic feature of the cut-rule is the existential quantifier in front of the interpretation. Observe that this quantifier has a very unexpected feature: the *uniqueness* of y . (*Proof:* Assume that $\vdash y, z'$ and $\vdash x, z' \in \pi'^*$, and that $\vdash y, z''$ and $\vdash x, z'' \in \pi''^*$; necessarily, $y \asymp x \text{ [mod } B]$ and $y \asymp x \text{ [mod } B^\perp]$ and necessarily, $y = x$.)

(iii) If π is obtained from π' by the exchange rule replacing A by a permutation $\sigma(A)$, then $\pi^* = \{\vdash \sigma(z) ; \vdash z \in \pi'^*\}$.

(iv) If π is the axiom $\vdash T, A$, then $\pi^* = \emptyset$.

(v) If π is obtained from π' and π'' by (&):

$$\frac{\vdash A, C \quad \vdash B, C}{\vdash A \& B, C},$$

then $\pi^* = \{\vdash (0, a), z' ; \vdash a, z' \in \pi'^*\} \cup \{\vdash (1, b), z'' ; \vdash b, z'' \in \pi''^*\}$.

(vi) If π is obtained from π' by (1 \oplus), then $\pi^* = \{\vdash (0, a), z' ; \vdash a, z' \in \pi'^*\}$.

(vii) If π is obtained from π' by (2 \oplus), then $\pi^* = \{\vdash (1, a), z' ; \vdash a, z' \in \pi'^*\}$.

(viii) If π is the axiom $\vdash 1$, then π^* consists of one point, namely $\vdash 1$.

(ix) If π comes from π' by the weakening rule (\perp):

$$\frac{\vdash A}{\vdash \perp, A},$$

then $\pi^* = \{\vdash \perp, z; \vdash z \in \pi'^*\}$.

(x) If π comes from π' and π'' by (\otimes):

$$\frac{\vdash A, C \quad \vdash B, D}{\vdash A \otimes B, C, D},$$

then $\pi^* = \{\vdash (z', z''), t'; \vdash z', t' \in \pi'^* \text{ and } \vdash z'', t'' \in \pi''^*\}$. Assume that $\vdash (z', z''), t'; t''$ and $\vdash (x', x''), w'; w''$ are two distinct points (u and v) or π^* so that, for instance, $\vdash z', t' \neq \vdash x', w'$; then,

- either $t' \cap w' \text{ [mod } C]$ in which case u and v are coherent,
- or $t' \asymp w' \text{ [mod } C]$ in which case $z' \cap x' \text{ [mod } A]$.

Now, if $z'' = x''$, we get the coherence of (z', z'') with (x', x'') , from which u and v are coherent; hence, we can assume that $\vdash z'', t'' \neq \vdash x'', w''$, in which case we have

- either $t'' \cap w'' \text{ [mod } D]$ and u and v are coherent,
- or $t'' \asymp w'' \text{ [mod } D]$ and so, $z'' \cap x'' \text{ [mod } B]$; but then, $(z', z'') \cap (x', x'') \text{ [mod } A \otimes B]$ and once more u and v are coherent.

(xi) If π comes from π' by (\wp):

$$\frac{\vdash A, B, C}{\vdash A \wp B, C},$$

then $\pi^* = \{\vdash (x, y), z; \vdash x, y, z \in \pi'^*\}$. In this case, there is practically nothing to verify.

(xii) If π is obtained from π' by weakening rule ($W?$):

$$\frac{\vdash B}{\vdash ?A, B},$$

then $\pi^* = \{\vdash \emptyset, z; \vdash z \in \pi'^*\}$.

(xiii) If π is obtained from π' by the dereliction rule ($D?$):

$$\frac{\vdash A, B}{\vdash ?A, B},$$

then $\pi^* = \{\vdash \{a\}, z; \vdash a, z \in \pi'^*\}$.

(xiv) If π is obtained from π' by the contraction rule ($C?$):

$$\frac{\vdash ?A, ?A, B}{\vdash ?A, B},$$

then $\pi^* = \{\vdash a \cup b, z; \vdash a, b, z \in \pi'^* \text{ and } a \cup b \in A^\perp\}$. The coherence of π^* is practically immediate.

(xv) If π is obtained from π' by the rule (!):

$$\frac{\vdash A, ?B}{\vdash !A, ?B},$$

then π^* is the set of all sequences of the form $\vdash \{x_1, \dots, x_n\}, z_1 \cup \dots \cup z_n$ such that

- (1) $\{x_1, \dots, x_n\} \in A$,
- (2) for $i = 1, \dots, n$, $\vdash x_i, z_i \in \pi'^*$,
- (3) $z_1 \cup \dots \cup z_n \in B^\perp$

(By the way, x_1, \dots, x_n are necessarily distinct: if $x_1 = x_2$, then $z_1 \cup z_2 \in B^\perp$ and the fact that the $\vdash x_i, z_i$'s are coherent shows that $z_1 = z_2$, in which case we have a useless repetition.)

The coherence of two elements $u = \{x_1, \dots, x_n\}, z_1 \cup \dots \cup z_n$ and $v = \{y_1, \dots, y_m\}, t_1 \cup \dots \cup t_m$ is shown as follows: assume $u \neq v$;

- if $\{x_1, \dots, x_n, y_1, \dots, y_m\} \in A$, then u and v are coherent;
- otherwise, one x_i (say x_1) is incoherent with one y_j (say y_1). But $\vdash x_1, z_1$ and $\vdash y_1, t_1$ are coherent; hence, $z_1 \cap t_1 \text{ [mod } ?B]$; this shows that $z_1 \cup \dots \cup z_n \cap t_1 \cup \dots \cup t_m \text{ [mod } ?B]$ and u and v are coherent.

(xvi) If π is obtained from π' by the rule (\wedge) :

$$\frac{\vdash A, B}{\vdash \wedge \alpha. A, B},$$

then $\pi^* = \{\vdash (X, a), z ; (X, a) \in |\wedge \alpha. A| \text{ and } \vdash a, z \in \pi'[X/\alpha]^*\}$. Here we use the substitution of the new constant X for α .

The compatibility of distinct sequences $u = (X, a), z$ and $v = (Y, b), t$ is shown as follows: take Z together with $f \in \text{COH}(X, Z)$, $g \in \text{COH}(Y, Z)$; then, consider $a' = A[f/\alpha](a)$ and $b' = A[g/\alpha](b)$. Then, the general properties of the interpretation, as studied in [5], show that $\vdash a', z$ and $\vdash b', t$ both belong to $\pi[Z/\alpha]^*$, hence, are coherent. Moreover, they are still distinct by the general properties of a trace. If z is coherent with t , then u is coherent with v ; otherwise, $a' \cap b' \text{ mod } A[Z/\alpha]$ and then, $(X, a) \cap (Y, b) \text{ [mod } \wedge \alpha. A]$ from which we conclude again.

(xvii) If π is obtained from π' by the rule (\vee) :

$$\frac{\vdash A[B/\alpha], C}{\vdash \vee \alpha. A, C},$$

then $\pi^* = \{\vdash (X, a), z ; (X, a) \in |\vee \alpha. A| \text{ and } \exists f \in \text{COH}(X, B) : \vdash A[f/\alpha](a), z \in \pi'^*\}$. Assume that $(X, a), z (= u)$ and $(Y, b), t (= v)$ are two distinct elements of π^* with witnesses f, g in $\text{COH}(X, B)$ and $\text{COH}(Y, B)$. If $a' = A[f/\alpha](a)$, if $b' = A[g/\alpha](b)$, then $a', z \cap b', t \text{ mod } A[B/\alpha], C$, and these two sequences are distinct. Then, either $z \cap t \text{ [mod } C]$ or $a' \cap b' \text{ mod } A[B/\alpha]$, in which case $(X, a) \cap (Y, b) \text{ [mod } \vee \alpha. A]$; in both cases, u is coherent with v . This rule is the other case (after the cut-rule) of an existential quantifier. Here too, we have *uniqueness*: if the element $(X, a), z$ of π^* is witnessed by both $f, g \in \text{COH}(X, A[B/\alpha])$, then, with $a' = A[f/\alpha](a)$, $a'' = A[g/\alpha](a)$, we get the coherence of a', z and a'', z . But, looking at the definition of $\vee \alpha. A$ we see that $a' \asymp a'' \text{ mod } A[B/\alpha]$, so $a' = a''$. Hence, the element a' 'above' a is uniquely determined (unfortunately, f is not uniquely determined because of the nonpreservation of kernels; however, the range of f is uniquely determined).

3.16. Remarks. (i) The fact that the cut-rule formally behaves like the existential rule can be explained as follows: Define a formula CUT as $\vee \alpha \cdot \alpha \otimes \alpha^\perp$; then, semantically speaking, CUT consists of a trivial unit coherent space with only one point, namely $(1, (1, 1)) (= a_0)$. Now it is easy to show that the cut-rule can be mimicked by the formula CUT:

$$\frac{\vdash A, B \quad \vdash A^\perp, C}{\vdash A \otimes A^\perp, B, C} \otimes \frac{\vdash A \otimes A^\perp, B, C}{\vdash \text{CUT}, B, C} \vee .$$

(This is not far from some techniques used in [6].) Now, given premises π' and π'' of a cut-rule, let us compare the semantic interpretation of the proofs π and μ obtained by applying in one case the cut-rule, in the other case the procedure given above: $\mu^* = \{\vdash a_0, z; z \in \pi^*\}$, which means that there is no significant difference between both interpretations, and the unicity features of the cut-rule are therefore explainable from the similar features of the existential rule.

(ii) Assume, just for a minute, that we try to compute the *semantics* of a proof π , i.e., we try to decide, given $z \in |A|$, whether or not $z \in \pi^*$. In all cases but (ii) and (xvii), the problem is immediately reduced to several simpler problems of the same type, with an effective bound on these problems. But in Definition 3.15(ii) and (xvii), the problem is reduced to infinitely many simpler ones, due to the presence of an unbounded existential quantifier. But this existential part is unique, i.e., in some sense there is a well-defined z' ‘above’ z , summarizing the existential requirements on z . We can also think of z' as *implicitly* defined by z . In particular, cut-elimination looks as the elimination of implicit features, an elimination which can be achieved when the formula proved has no existential type.

This basic remark is the key to a semantic approach to computation, which will be undertaken somewhere else. Also observe that we are tempted to remove not only cuts, but also (\vee)-rules!

The just given semantics of sequential proofs induces a semantics of the corresponding proof-nets: one simply has to make the boring verification that, whenever $\pi^- = \mu^-$, we have $\pi^* = \mu^*$. Now, one could dream to work with proof-nets directly, without sequentializing. We shall illustrate this by a semantic interpretation of PN0, directly defined on the proof-nets. PN0 has been chosen because of its syntactic simplicity, and because in PN0 all the difficulties of the task are concentrated; from the sketch given here, there is no problem to move to PN2.

3.17. Definition (experiments). Let β be a proof-net in PN0. For each terminal formula A of β , we select an element a from the web of (the interpretation of) A . Now, by moving upwards, we construct, for each formula of the net, say B , an element $b \in |B|$. For the symmetry of the argument, it is convenient to consider the terminal symbol CUT as interpreted as a unit coherent space $\{a_0\}$.

(i) Assume that we have a binary \otimes - or \wp -link $\frac{A}{C} \otimes B$; we have already chosen $c \in |C|$; now, $|C| = |A| \otimes |B|$, i.e., $c = (a, b)$; then we choose a in $|A|$, b in $|B|$.

(ii) Assume that we have a cut-link $\frac{A}{\text{CUT}} A^\perp$; in $|\text{CUT}|$, a_0 has been chosen by convention. We select an arbitrary element $a \in |A|$ (if $|A| \neq \emptyset$) and we put this element a both in $|A|$ and $|A^\perp|$; in case $|A| = \emptyset$, the process fails.

By moving upwards, we eventually obtain (provided we never fail) an assignment of points for each formula of the net and this assignment is called an *experiment*. In the experiment, some choices have been made which were not mechanical. Now, for each axiom link $A \rightarrow A^\perp$, we have selected points $a' \in |A|$, $a'' \in |A|$. The experiment *succeeds* when, for any such link, the two choices are the same, i.e., $a' = a''$.

A sequence of points in the interpretations of the conclusion of β belongs to β^* exactly when there is a successful experiment starting with those points.

3.18. Compatibility Theorem. *Assume that we have made two experiments in β (corresponding to two terminal sequences a' , a'') and that these experiments succeed; then a' and a'' are coherent (in the sense of the par of the conclusions).*

Proof. Immediate consequence of the following lemma.

3.18.1. Compatibility Lemma. *Assume that b' and b'' are two experiments in β , and that, for any conclusion A of β , the corresponding points a', a'' are incoherent: $a' \not\asymp a'' \pmod{A}$; then the two experiments coincide. (In particular, there is exactly one successful experiment corresponding to a given sequence of β^* .)*

Proof. For each formula A of B , we shall write $A : \subset$ (respectively, \cap , \asymp , \cup , $=$, \neq) to indicate the corresponding relation between the two inhabitants of A that have been chosen in both experiments. Now, the proof proceeds as follows.

(i) In the case of a \otimes -link $\frac{A}{A \otimes B} B$, we shall distinguish between two cases:

Case 1: $A \otimes B : \subset$ or $A : \cup$;

Case 2: $A \otimes B : \subset$ or $B : \cup$.

Clearly, one of these cases holds.

(ii) In the case of a \wp -link $\frac{A}{A \wp B} B$, we shall distinguish between two cases:

Case 1: $A \wp B : =$ or $A : \neq$;

Case 2: $A \wp B : =$ or $B : \neq$.

Here too, one of these cases holds.

(iii) We shall now prove the impossibility of being always in Case 1, except if $b' = b''$. In fact, if n is the number of multiplicative switches, there are 2^n simultaneous cases that may occur, and, by symmetry, the argument made in this particular situation immediately extends. From this Lemma 3.18.1 will follow.

(iv) We shall make a trip starting with a given terminal formula A_1 . The hypothesis is $A_1 : \asymp$. At the end of the trip, we want to arrive with the conclusion $A_1 : \subset$. We shall visit all formulas twice: upwards, we shall conclude $A : \asymp$; downwards, we

shall conclude $A:\square$. Hence, a formula A visited twice is such that $A:=$, and we shall be done. When we start, the switches are not yet set; this will be done as follows:

(1) Assume that we arrive for the *first* time in a link $\frac{A}{A \otimes B}^B$, and that we arrive through the conclusion $A \otimes B$; so we are upwards, i.e., $A \otimes B:\asymp$. Since we are in Case 1, $A:\asymp$, so set the *switch on* “R” to go to A^\wedge .

(2) Assume that we arrive for the *first* time in a link $\frac{A}{A \otimes B}^B$, and that we arrive through the premise A ; so we are downwards, i.e., $A:\square$. Since we are in Case 1, $A \otimes B:\square$, so set the *switch on* “L” to move to $A \otimes B_v$.

(3) As in (2), but arriving through B : *switch on* “R”.

(4) Assume that we arrive for the *first* time in a link $\frac{A}{A \wp B}^B$, and that we arrive through the conclusion $A \wp B$; so we are upwards, i.e., $A \wp B:\asymp$. But then, observe that $B:\asymp$, so set the *switch on* “R” to move to B^\wedge .

(5) Assume that we arrive for the *first* time in a link $\frac{A}{A \wp B}^B$, and that we arrive through the premise A ; so we are downwards, i.e., $A:\square$. Since we are in Case 1, $A \wp B:\square$, so set the *switch on* “L” to move to $A \wp B_v$.

(6) As in (5), but arriving through B : we are downwards, i.e., $B:\square$; we have two subcases:

(6)(a): if $B:=$, then *switch on* “L” to go to B^\wedge ;

(6)(b): if $B\neq$, then $A \wp B:\frown$; hence, $A \wp B:\square$; *switch on* “R” to go to $A \wp B_v$.

(7) Assume that we arrive for the *second* time in a link $\frac{A}{A \otimes B}^B$ and that the first passage was as in (1); then we come back through A_v , i.e., $A:\square$. Since we are in Case 1, $A \otimes B:\square$ and since we had $A \otimes B:\asymp$, we get $A \otimes B:=$. But then, $B:\square$ and the next step to B^\wedge will be sound.

(8) Assume that we arrive for the *second* time in a link $\frac{A}{A \otimes B}^B$, and that the first time was as in (2) or (3); now, the order of passage is such that we arrive through $A \otimes B^\wedge$, i.e., $A \otimes B:\asymp$; but we already had $A \otimes B:\square$, so $A \otimes B:=$ and so, $A:\asymp$, $B:\asymp$ and the next step to A^\wedge or B^\wedge will be sound.

(9) Assume that we arrive for the *second* time in a link $\frac{A}{A \wp B}^B$, and that the first time was as in (4); so we arrive through A_v , i.e., $A:\square$; but we had $A \wp B:\asymp$ from which $A:\asymp$, so $A:=$, but we are in Case 1, hence, $A \wp B:=$. The next step of the trip to A^\wedge is clearly sound since $A:\square$.

(10) Assume that we arrive for the *second* time in a link $\frac{A}{A \wp B}^B$, and that the first passage was as in (5) or (6); we have two subcases:

(10)(a): first passage as in (6)(a): we arrive through A_v , i.e., $A:\square$, but we had $B:=$, hence, $A \wp B:\square$ and the next step to $A \wp B$ will be sound;

(10)(b): first passage as in (5) or (6)(b): we arrive through $A \wp B^\wedge$ and we had already passed through $A \wp B_v$, so $A \wp B:=$, hence, the next step to A^\wedge or B^\wedge will be sound, since $A:=$, $B:=$.

(11) Assume that we arrive for the *third* time in a multiplicative link $\frac{A}{C}^B$; we have already seen in cases (7), (8), (9) and (10)(b) that $C:=$. In case (10)(a) for the second passage, we are in $A \wp B^\wedge$ after being in $A \wp B_v$, so here too, $C:=$; but then, in every case, $A:=$, $B:=$, and the next step is sound.

(12) Assume that we arrive in a *conclusion* A_i , i.e., $A_i : \Box$; then we can move to A_i^\wedge because of the hypothesis $A_i : \asymp$.

(13) Assume that we arrive on one side of a cut-rule $\frac{A}{\text{CUT}} \frac{A^\perp}{A}$, say through A_v ; then, $A : \Box$ from which it is immediate that $A^\perp : \asymp$ (the points are the same) and so the move to A_v^\perp is logically sound.

(14) Assume that we arrive on one side of an axiom $\frac{A}{A^\perp}$, say through A^\wedge ; then, $A : \asymp$ from which it is immediate that $A^\perp : \Box$ (the points are the same) and the move to A_v^\perp is logically sound.

(v) Summing up now, we have obtained $A :=$ for any formula A . \square

3.19. Remark. The same argument works for PN2, with a heavier apparatus. In experiments, we have to choose existential witnesses, but the same unicity can be obtained; i.e., there is only one successful experiment. (In a rule

$$\frac{A[B/a]}{\vee a.A}$$

observe that $\vee a.A : \asymp$ implies $A[B/a] : \asymp$, and that $A[B/a] : \Box$ implies $\vee a.A : \Box$.)

4. Normalization in PN2

We shall define a normalizing algorithm for proof-nets. The algorithm is a rewriting procedure (*contraction*) whose properties are as follows:

- (i) If $\beta \text{ cntr } \beta'$, then β and β' have the same terminal formulas (conclusions).
- (ii) If $\beta \text{ cntr } \beta'$, then $\beta'^* = \beta^*$ (semantic soundness).
- (iii) There is a β' such that $\beta \text{ cntr } \beta'$ iff β contains CUT-links; a proof-net β with no CUT-link is said to be *normal*.

Reduction is defined as the transitive closure of *cntr*, and is denoted $B =/ \beta'$. It is sometimes useful to speak of a *reduction sequence* from β to β' , i.e.,

$$\beta = \beta_0 \text{ cntr } \beta_1 \text{ cntr } \cdots \beta_{n-1} \text{ cntr } \beta_n = \beta'.$$

We can also use the notation $\beta =/_n \beta'$ to indicate the existence of a reduction sequence of length $n+1$ from β to β' . The main property of reduction is the following:

- (iv) *Strong normalization*: A proof-net β is said to be SN when there is no infinite reduction sequence starting from β ; in this case we define the integer $N(\beta)$ as the greatest n such that $\beta =/_n \beta'$ for some β' . (By König's Lemma, using the fact that, for a given β , there are finitely many β' such that $\beta =/ \beta'$, $N(\beta)$ is definable.) It may be useful to define $N(\beta)$ in general by

$$N(\beta) = \{\sup n ; \exists \beta' (\beta =/_n \beta')\}$$

so that β is SN exactly when $N(\beta) < \omega$. The Strong Normalization Theorem, which is the main result of this section, says that all proof-nets of PN2 are SN. One of

the main problems with our procedure is that it is not Church-Rosser, i.e., β may reduce to *normal forms* β' and β'' which are distinct. However, $\beta^* = \beta''^*$ so that they do not differ too much. In particular we shall see in Section V how to represent booleans, integers, etc. in PN2.

A given integer (say 10) has several representations in PN2, all very close one to another. What we want is that if one reduction sequence yields the result 10, another does not yield the result 28; semantically speaking however, any representation of 10 in PN2 is interpreted in the same way as some set 10^* , similarly for 28 and, clearly, $10^* \neq 28^*$; hence there is no ambiguity.

One of the most interesting features of strong normalization is the possibility to ignore certain normalization rules. If we decide to use all contractions except, say (R) , we eventually reach a normal form relative to (R) , i.e., a proof-net where the only contractions that can be performed are instances of (R) . In some cases, this is enough to ensure that we reach a normal form without using (R) .

Let us give an example demonstrating this possibility: Imagine that we ignore the commutation rule for $\&$. If we start with a proof-net not involving $\&$ or \vee in its conclusions, and take a relative normal form β for it, then assume β is not normal. We start with a CUT $\frac{C}{\text{CUT}} \frac{C^\perp}{\text{CUT}}$, where C^\perp is the auxiliary door of a $\&$ -box. So let us go to the front-door C' of this box: the downmost formula below C' , say D , bears one of the symbols $\&$ or \vee , so it cannot be a conclusion, and is therefore the premise of a CUT-rule. The other premise D^\perp is again the auxiliary door of a $\&$ -box. Iterating the process we eventually find a cycle since there are finitely many cuts in β from which a shorttrip is easily made. Thus, we have contradicted the assumption that β was not cut-free.

By the way, observe that it was also possible in that case to forbid contractions inside $\&$ -boxes and still arrive at a normal form: strong normalization justifies *lazy* strategies for computation.

We shall now describe the normalization procedure by giving the contraction rules. They are divided into three groups, *axiom* contraction (Section 4.1) *symmetric* contractions (Section 4.2), and *commutative* contractions (Section 4.3). For each of these groups we establish, with the definition, the semantic soundness of the contraction rule. In order to save space, it is very convenient to consider the CUT-link as symmetric, i.e., $\frac{C}{\text{CUT}} \frac{C^\perp}{\text{CUT}}$ and $\frac{C^\perp}{\text{CUT}} \frac{C}{\text{CUT}}$ are exactly the same link. In particular, in our graphical representations, it will be possible to put what is most convenient on the left, in order to avoid duplicating all cases!

4.1. Axiom contraction

The axiom contraction (AC) can be performed every time one of the premises of a cut-link is the conclusion of an axiom link: it consists in replacing a configuration:

$$\frac{\overline{A} \quad A^\perp}{\text{CUT}} \quad \downarrow \quad A \quad \text{by:} \quad \downarrow \quad A,$$

i.e., by identifying the two occurrences of A and removing A^\perp and the CUT- and axiom-links. We have already seen that the two occurrences of A are necessarily distinct.

We show the semantic soundness of (AC): assume that $\beta \text{ cntr } \beta'$ by (AC), then $\beta^* = \beta'^*$ is shown by induction on the number n of boxes containing the CUT we contract.

Basis step: If $n = 0$, then consider an experiment in β' corresponding to a certain choice of values z for the conclusions. If the experiment succeeds, let x be the point chosen in the formula A . Now, we can form an experiment in β corresponding to the same initial choice z : for the cut-rule, choose x , both in A and A^\perp , and report the values from the given experiment everywhere else. This new experiment succeeds, so $\beta^* \subset \beta'^*$. Conversely, from a successful experiment in β , it is not difficult to deduce a successful experiment in β' : at the level of the cut-rule, we have to guess a value $y \in |A| = |A^\perp|$, but, in order for the experiment to succeed, this value has to be the same as the value x already chosen in the 'left' A , and so it is perfectly harmless to identify the two A 's: $\beta^* \subset \beta'^*$.

Induction step: If $n \neq 0$, then the contraction takes place within a box \mathbb{B} , replaced by \mathbb{B}' . Now, boxes are built in such a way that (using the induction hypothesis) $\mathbb{B}^* = \mathbb{B}'^*$. Then the semantics of β and β' will coincide since they are built in the same way from these boxes. (In later contractions, as this argument is perfectly general, we shall always assume that the contraction is not done within a box.)

4.2. Symmetric contractions

This concerns all CUT-links where one side comes by a rule, whereas the other side comes by the dual rule.

4.1. Definition (T -contraction). Since there is no rule for 0 , this case never occurs!

4.2. Definition (addition contraction: $(\&/1\oplus\text{-SC})$). A configuration as in Fig. 18(a) is replaced by the one in Fig. 18(b).

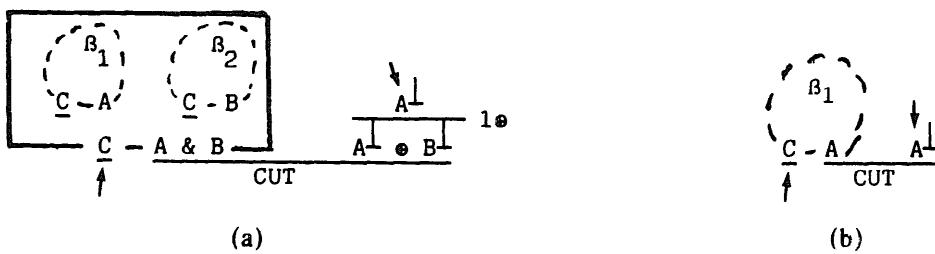


Fig. 18. $(\&/1 \oplus\text{-SC})$.

The semantic soundness is shown as follows: In β , at some moment, we have $c \in |C|$ and we have to guess an element $(0, a)$ or $(1, b)$ of $|A \& B|$. Inside the box, one goes on with c, a in β_1 or c, b in β_2 , according to the case. However, due to

the $(1\oplus)$ -rule, the choice of $(1, b)$ would be a failure, so the only successful choice is $(0, a)$, in which case we proceed above A^\perp by choosing a . Now, in β' , we have to choose some a in $|A|$, and then go up in β_1 and in A^\perp . This establishes clearly that $\beta^* = \beta'^*$.

4.3. Definition (additive contraction: $(\&/2\oplus\text{-SC})$). This case is perfectly symmetric to $(\&/1\oplus\text{-SC})$.

4.4. Definition (unit contraction: $(1/\perp\text{-SC})$). A configuration as in Fig. 19(a) is replaced by the one in Fig. 19(b).

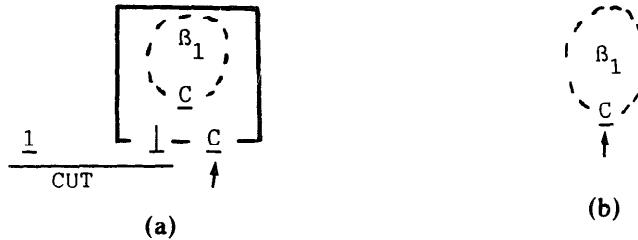


Fig. 19. $(1/\perp\text{-SC})$.

The semantic soundness is immediate: In β , we have to guess an element of $|1|$, and there is only one choice: 1. This value is reported in the box, but not used. What matters is the choice of values $c \in C$, just as in β' , so $\beta^* = \beta'^*$.

4.5. Definition (multiplicative contraction: $(\otimes/\wp\text{-SC})$). A configuration as in Fig. 20(a) is replaced by the one in Fig. 20(b).

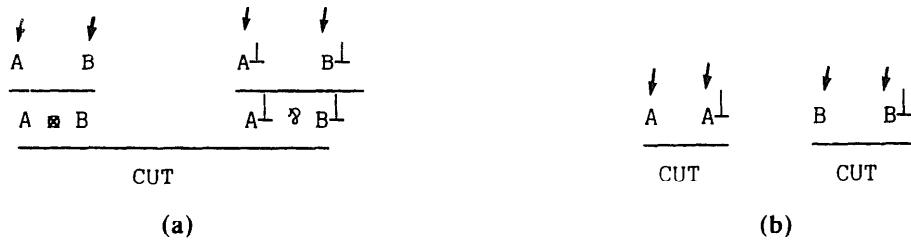
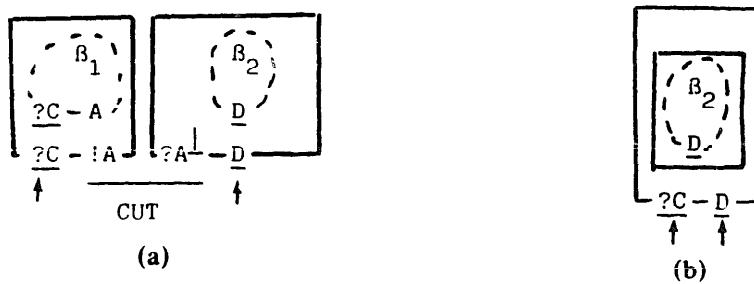


Fig. 20. $(\otimes/\wp\text{-SC})$.

The semantic soundness is proved as follows: In β we have to guess a pair (x, y) , then report it in the two premises, and then dispatch the components, x in A and A^\perp , y in B and B^\perp . But this amounts exactly to guessing (in β') x in $|A|$ and y in $|B|$, etc.

4.6. Definition (exponential contraction $(!/\text{W?}-\text{SC})$). A configuration as in Fig. 21(a) is replaced by the one in Fig. 21(b), where several weakenings are performed. This is a typical example where Church-Rosser is violated. Now, if we were allowed to put several weakenings in the same box (what any reasonable implementation should do), the problem would at least disappear from this case.

Fig. 21. ($!/\mathbf{W}?\text{-SC}$).

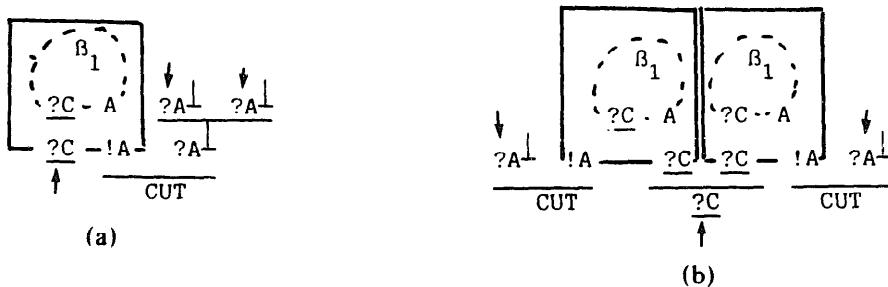
The semantic soundness is proved as follows: In β , we are given sequences c (in $?C$) and d (in D). We have to guess an element a of $!A$. Now, the element has to be transferred into the right box, and the general definition says that $a = \emptyset$. Also, if $a = \emptyset$, the general definition for the left box imposes the requirement that all elements of the sequence c are void, nothing else. Hence, in β , we only require $c = \emptyset$ and then transfer d to β_2 , which is exactly what we do in β' .

4.7. Definition (exponential contraction ($!/\mathbf{D}?\text{-SC}$)). A configuration as in Fig. 22(a) is replaced by the one in Fig. 22(b).

Fig. 22. ($!/\mathbf{D}?\text{-SC}$).

The semantic soundness is established as follows: In β , given the sequence c in $?C$, we have to guess a point $a \in !A$. Now, the dereliction rule imposes that a is a singleton $\{z\}$, and we proceed with z in A^\perp . But, if we enter the box with c and $\{z\}$, this means that c, z are reported inside, as can be seen from the ‘uniqueness remark’ in Definition 3.15(xv). This means that, in fact, we guess a point z in $|A|$, and proceed as in β' .

4.8. Definition (exponential contraction ($!/\mathbf{C}?\text{-SC}$)). A configuration as in Fig. 23(a) is replaced by the one in Fig. 23(b). In this contraction, there is a duplication of a

Fig. 23. ($!/\mathbf{C}?\text{-SC}$).

box, and several rules ($C?$) are used; this rule is responsible for the loss of control over the normalization time.

The semantic soundness of the rule is established as follows: In β , we start with c in $?C$, and we guess some a in $!A$. For the rule ($C?$), we have to guess again a decomposition $a = a' \cup a''$. We enter the box with c, a . In β' , we have to guess elements a' (left) and a'' (right) of $|A|$. But we also have to guess a decomposition of c as $c' \cup c''$. Then we have two boxes to enter: one with c', a' and one with c'', a'' . Now, the box \mathbb{B} , which is used in β and recopied twice in β' , is such that if $c', a'/c'', a''$ belong to \mathbb{B}^* , so do $c' \cup c''$ and $a' \cup a''$. Conversely, if $c, a' \cup a'' \in \mathbb{B}^*$, then c can be uniquely decomposed as $c' \cup c''$ so that $c', a'/c'', a'' \in \mathbb{B}^*$. This means that the successful experiments in both proof-nets are in 1-1 correspondence and we are done.

4.9. Definition (quantification contraction (\wedge/\vee -SC)). Replace a configuration as in Fig. 24(a) by the one in Fig. 24(b).

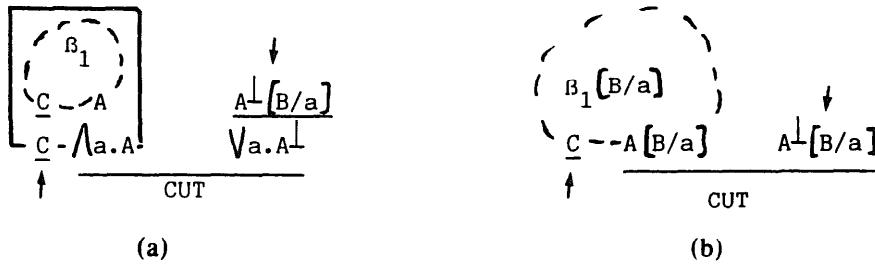


Fig. 24. (\wedge/\vee -SC).

The semantic soundness is checked as follows: We start with c in C and we have to guess an element (X, z) in $|\wedge a.A|$. Above the rule (\vee), we have to guess an element $f \in \text{COH}(X, B)$ in order to get $z' = A[f/a]^+(z)$, a point in $|A[B/a]|$. We also have to check that c and (X, z) are in \mathbb{B}^* , where \mathbb{B} is the left box. Now, the general functoriality properties (see [5]) make it the same as to verify that c, z' is in $\beta_1[B/a]^*$. From this, the soundness of this case follows.

4.3. Commutative contractions

4.10. Definition. In order to state the commutative contractions, we need the concept of a *ghost box*. The ghost boxes can be formed for CUT- or \otimes -links (this is just a geometrical feature), and we discuss them for a CUT-link only. So, if we have such a link $\frac{B}{\text{CUT}} \frac{B^\perp}{\text{CUT}}$, we can consider the empire eB of B . In eB , there is a *frontier*, made of B , of the conclusions of β belonging to eB (in this description, we assume that the CUT is not in a box, just to simplify everything), and of premises of (\wp)- and ($C?$)-links such that the other premise is not in the box. Let us list the frontier of eB as A, B ; we can form a box (the ghost box) by putting eB inside, and the doors of the box will be A, B . If we replace eB in β by this ghost box, we are executing a *materialization*.

4.11. Remarks. (i) In terms of computation, materialization is a very feasible process since it suffices to start a trip with B^\wedge and to set the switches arbitrarily. However, in the case we enter a rule (\wp) or ($C?$) for the first time and from above, the switch is set such that we move backwards: when we eventually arrive in B_v , we have passed exactly through eB , so there is no problem to describe eB very quickly.

(ii) When, in turn, our CUT is in a box, then the box is made out of one of several proof-nets arranged together, and the CUT occurs in one of these nets; in this net, the concept of a ghost box makes sense, and it is in this way that we define the ghost box of a nested cut.

(iii) The main point is to prove that *materialization* produces a sound proof-net; in fact, we shall establish more, namely that we can simultaneously materialize eB and eB^\perp as is stated in the following theorem.

4.12. Theorem. *The proof-structure obtained by materializing eB and eB^\perp is sound.*

Proof. We argue by induction on the number of links outside $eB \cup eB^\perp$. In the case of PN0 all the difficulties are concentrated, hence, we restrict ourselves to this case.

(i) If the only link outside $eB \cup eB^\perp$ is the CUT-link, then the proof splits and the materialization looks as shown in Fig. 25, which is a proof-net.

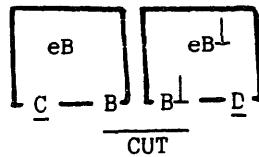


Fig. 25.

(ii) If there is a terminal \wp -link outside $eB \cup eB^\perp$, then remove it in order to get β' . Now, the materialization of β is the materialization of β' to which we have added the terminal \wp -link.

(iii) Otherwise, observe that there is a terminal \otimes - or CUT-link with premises E and F such that $eE \cup eF$ is maximal and $eB \cup eB^\perp \subset eE$ (or eF). The reason for this is that, when we have a terminal \otimes - or CUT-link such that the empires eC and eD above are not such that $eC \cup eD$ is maximal, then we can conclude the existence of a conflicting \wp -link, as in the Splitting Theorem; now, what is below this conflict is in neither empires; hence, if all terminal \wp -links are in $eC \cup eD$, it follows that there is a terminal \otimes - or CUT-link below the conflict. We can conclude as in the Splitting Theorem that $eC \cup eD \subset eE$ (or eF). Now, there is a splitting of β as shown in Fig. 26 (case of a tensor-link, $eC \cup eD \subset eE$). From the materialization in β_1 , it is easy to obtain the materialization in β . \square

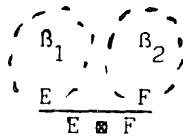


Fig. 26.

It remains to examine the CUT-links which are not covered by the two cases already considered in the proof of Theorem 4.12. The only possibility is that at least one premise is the auxiliary door of a box.

4.13. Definition (zero commutation (\top -CC)). The configuration in Fig. 27(a) is replaced by the one in Fig. 27(b). To do this, we have proceeded as follows: In a

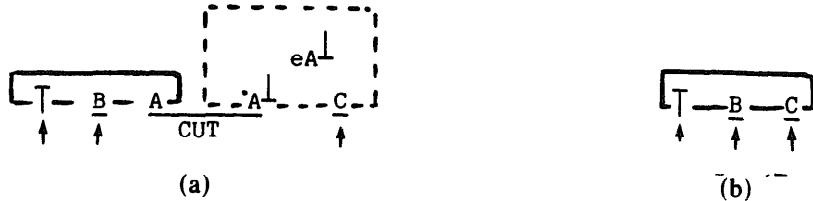


Fig. 27. (\top -CC).

first time, we have a materialization of the ghost box eA^\perp , in a second time, we have the replacement of the configuration involving both boxes by another one.

The semantic soundness is vacuously satisfied since, due to the presence of \top with its void web, in both cases we never have anything to verify at this stage.

4.14. Definition (additive commutation (&-CC)). The configuration in Fig. 28(a) is replaced by the one in Fig. 28(b).

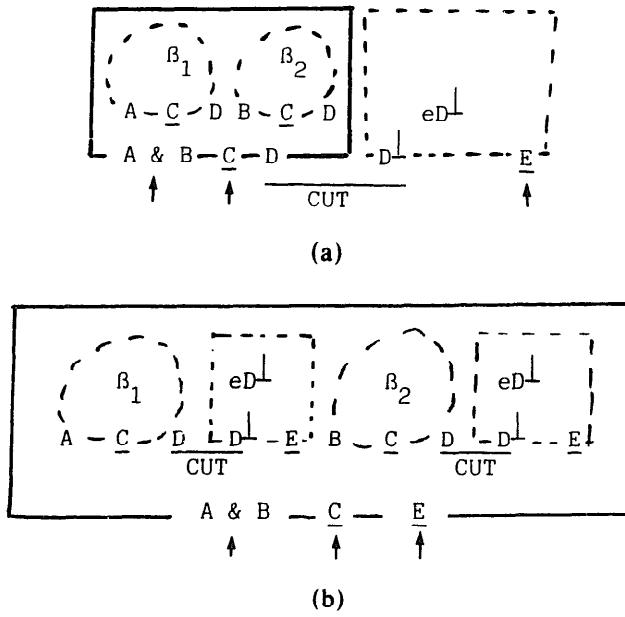


Fig. 28. (&-CC).

The semantic soundness is established as follows: We are given c, e in C, E and a point in $|A \& B|$, say $(0, a)$. In β we have to guess $d \in |D|$; from this we proceed by putting $(0, a), c, d$ in the left box and after a, c, d in β_1 , we also put d, e in the ghost box. In β' , we proceed in a different order, namely that $(0, a), c, e$ is transferred

to the left box and then looks as a, c, e , inside the box. Then we have to guess $d \in |D|$ and then dispatch a, c, d in β_1 , d, e in eD^\perp .

Remark: Of course, the ghost box is used only temporarily: after writing the contractum, one erases it; the fact that the erasing of a ghost box causes no problem is practically immediate.

4.15. Definition (unit commutation (\perp -CC)). The configuration in Fig. 29(a) is replaced by the one in Fig. 29(b).

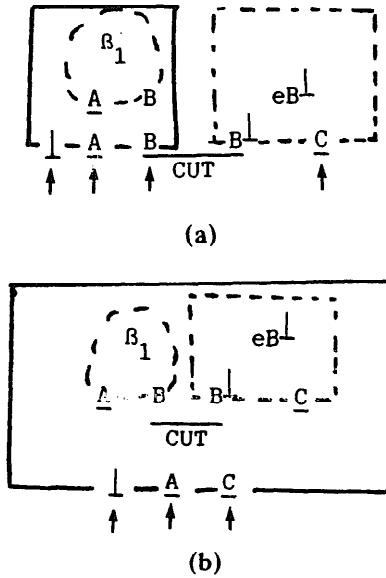


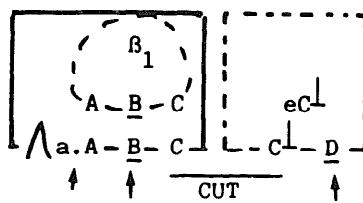
Fig. 29. (\perp -CC).

The semantic soundness of this contraction is established as follows: In β , we start with $1 \in |\perp|$, a in A , and $c \in |C|$. We then have to guess some $b \in |B|$. Once b has been chosen, these elements are dispatched between the two boxes: b, c in the ghost box, $1, a, b$ in the left box, i.e., a, b in β_1 . What happens in β is almost the same: $1, a, b$ is transferred to the box so that it becomes a, C ; then we guess d , and a, b go to β_1 while b, c go to eB .

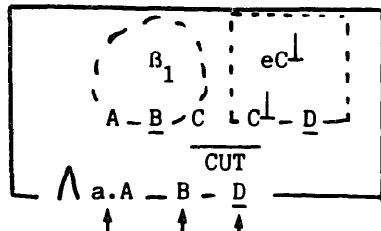
4.16. Definition (exponential commutation (W?-CC)). This case is exactly the same as the one in Definition 4.15 except that the formula \perp is replaced by a formula $?D$.

4.17. Definition (quantificative commutation (\wedge -CC)). The configuration in Fig. 30(a) is replaced by the one in Fig. 30(b).

The soundness is proved as follows: In β , we start with $(X, z) \in |\wedge a.A|$, b in B , and d in D ; we guess $c \in |C|$. From this, we dispatch c, d to eC^\perp and $(X, z), b, c$ to β_1 . This means that z, b, c is transferred to $\beta_1[X/a]$. In β' , starting with the same data, we transfer them to the interior of the box and they become z, b, d , but β_1 has been replaced by $\beta_1[X/a]$. Then we guess $c \in |C|$, and z, b, c is transferred to $\beta_1[X/a]$ while c, d goes to eC^\perp .



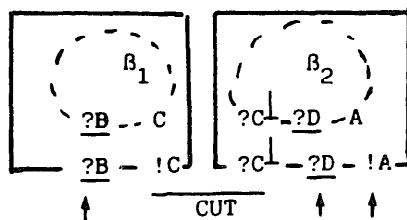
(a)



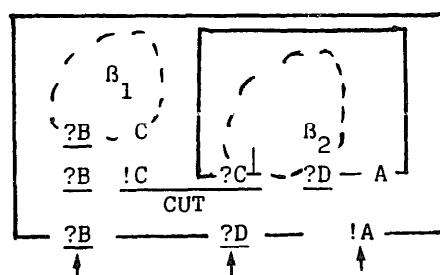
(b)

Fig. 30. (\wedge -CC).

4.18. Definition (exponential commutation (!-CC)). The last case is that of a !-box. Here, the ghost boxes would be of no use, so there is no general pattern of commutation. However, we must be able to do something in all possible CUT-situations. Let us look, however, at the other premise of the CUT: it is of the form $!C$. Now, if we cannot do something on that side, this means that $!C$ is neither the conclusion of an axiom-link nor the auxiliary door of a box; the only possibility is that $!C$ is the main door of a !-box: the configuration in Fig. 31(a) is replaced by the one in Fig. 31(b).



(a)



(b)

Fig. 31. (!-CC).

In order to prove the soundness, consider b, d, a in $?B, ?D, ?A$; we first have to guess (in β) an element $c \in |C|$, and then to dispatch it between the two boxes: on the left, it becomes b, c on the right c, d, a . Assume that $a = \{x_1, \dots, x_n\}$, $c = \{z_1, \dots, z_m\}$; within the boxes, we have to guess decompositions $b = b_1 \cup \dots \cup b_m$, and then consider simultaneously $b_1, z_1 / \dots / b_m, z_m$. Further, $c = c_1 \cup \dots \cup c_n, d = d_1 \cup \dots \cup d_n$ and we have to consider simultaneously $c_1, d_1, x_1 / \dots / c_n, d_n, x_n$. Now, consider b'_1, \dots, b'_n defined by $b'_i = \bigcup \{b_j ; z_j \in c_i\}$. Then the left part of the experiment amounts to the same as verifying that $b'_1, c_1 / \dots / b'_n, c_n$ belong to the left box. Now, if we start with the same initial data in β' , we are first faced with guessing a decomposition $b = b'_1 \cup \dots \cup b'_n$ and $d = d_1 \cup \dots \cup d_n$ followed by separately studying $b_1, d_1, x_1 / \dots / b_n, d_n, x_n$. In the case of t_i, d_i, x_i , we have to guess some c_i in $|C|$ and then dispatch between the two sides b_i, c_i on the left, c_i, d_i, x_i on the right, and the problem is therefore strictly equivalent to the problem of β .

4.19. Remark. Any binary link with the same geometry as CUT, typically \otimes , can give rise to commutative contractions; in particular, the analogues of Definitions 4.13–4.18 for \otimes are easily written.

Our goal is, as stated, *strong normalization*. It is however quite interesting to give the *Small Normalization Theorem* first, which gives a very quick normalization proof for PN2, provided $(!/C?-SC)$ is not used.

4.20. Definition. The *size* $s(\beta)$ of a proof-net β is defined by induction on β as follows:

- (i) If β is built from boxes B_1, \dots, B_n and formulas A_1, \dots, A_k by means of usual unary or binary links, then $s(\beta) = s(B_1) + \dots + s(B_n) + 2^k$ (in other terms, formulas are counted to be of size 2).
- (ii) If β is a box, then the size of β is defined as follows:
 - if β is a T-box, T, C_1, \dots, C_n , then $s(\beta) = 2 + n$;
 - if β comes from β' by the schemes for \perp , $W?$, \wedge , $!$, then $s(\beta) = s(\beta') + 1$;
 - if β comes from β', β'' by the scheme for $\&$, then $s(\beta) = s(\beta') + s(\beta'') + 1$.

4.21. Lemma. *In all contraction rules but $(!/C?-SC)$, the size strictly diminishes.*

Proof. We have to go through 14 cases. First, observe that $s(\beta) > n$ if β has n conclusions. In order to prove the lemma, one can make the simplifying hypothesis that the cut contracted is not inside a box.

- (AC) divides the size by 4;
- in $(\&/1\oplus\text{-SC})$, a size $(x+y+1) \cdot 2z$ is replaced by a size $x \cdot z$;
- in $(1/\perp\text{-SC})$, a size $2 \cdot (x+1)$ is replaced by a size x ;
- in $(\otimes/\wp\text{-SC})$, a size $x \cdot 2^6$ is replaced by a size $x \cdot 2^4$;
- in $(!/W?\text{-SC})$, a size $(x+1) \cdot (y+1) \cdot z$ is replaced by a size $(y+n) \cdot z$ for some $n \leq x$;

- in $(!/\mathcal{D}?\text{-SC})$, a size $(x+1) \cdot 4z$ is replaced by a size $x \cdot 2z$;
- in $(\wedge/\vee\text{-SC})$, a size $(x+1) \cdot 4z$ is replaced by a size $x \cdot 2z$;
- in $(\top\text{-CC})$, a size $(3+n) \cdot x \cdot y$ is replaced by a size $(2+n+m) \cdot y$, with $m < x$;
- in $(\&\text{-CC})$, a size $(x+y+1) \cdot z \cdot t$ is replaced by a size $(xz+yz+1) \cdot t$;
- in $(\perp\text{-CC})$, a size $(x+1) \cdot y \cdot z$ is replaced by a size $(xy+1) \cdot z$;
- in $(\wedge\text{-CC})$, a size $(x+1) \cdot y \cdot z$ is replaced by a size $(xy+1) \cdot z$;
- in $(!\text{-CC})$, a size $(x+1) \cdot (y+1) \cdot z$ is replaced by a size $(x \cdot (y+1)+1) \cdot z$.

All cases have been treated except trivial variants. \square

4.22. Corollary (Small Normalization Theorem). PN2 without $(!/\mathcal{C}?\text{-SC})$ satisfies strong normalization.

Proof. Clearly, $N(\beta) \leq s(\beta)$. \square

4.23. Remarks. (i) In $(!/\mathcal{C}?\text{-SC})$, a size $(x+1) \cdot 8z$ becomes $(x+1)^2 \cdot 4z \cdot 2^n$, where $n < x$. Suddenly, the size explodes. Is it possible to fix this defect? Obviously not, because PN2 has, roughly spoken, the same expressive power as the system F , i.e. the normalization theorem for F is not provable in PA_2 , so it would simply be childish to try to improve the Small Normalization Theorem by toying with the definition. There is however an open possibility: one can define generalized sizes (no longer integers), typically in the case of the !-box, so that we still have a phenomenon of decrease; these generalized numbers could be something like ordinals, dilators, ptykes. Such an achievement seems highly probable (but difficult); it would immediately yield something like an ordinal analysis of PA_2 .

(ii) To some extent, the Small Normalization Theorem is more important than its more refined and delicate elder brother! This is because normalization is now clearly divided into two aspects.

(1) The quick normalization devices which reduce size: These devices must be seen as *communication devices* and our small theorem says that this part works wonderfully well and quickly in PN2 (for those who may be afraid by the exponentials involved in sizes, remember that there is no necessity, no intention to work sequentially...). In particular, we could have put nonterminating devices inside those awful !-boxes; this would not affect the quality of the parallelism, the programs would simply run forever. But the structure Q/A would remain sound.

(2) The device $(!/\mathcal{C}\text{-SC})$, which can be seen as the *execution device* of the system: The Strong Normalization Theorem will prove that this device, together with the others, leads to terminating algorithms, and this is essential for the expressive power of PN2 which should be essentially the same as that of F . General parallel systems, for instance those that run forever, could be obtained by taking different execution devices, to be put in carefully sealed boxes!

In spite of the defect w.r.t. the Church-Rosser property, the interconvertibility w.r.t. the communication devices could be seen as a serious candidate for expressing equivalence of programs.

We now turn our attention towards the general case; for this we need to establish some facts about strong normalization relating it to simple normalization.

4.24. Definition. A contraction is *standard* when it does not erase any symbol CUT besides the one explicitly considered. Concretely, this means that some parts of the configuration we replace have to be cut-free, namely:

- in $(\&/1\oplus\text{-SC})$, β_2 must be cut-free,
- in $(\&/2\oplus\text{-SC})$, β_1 must be cut-free,
- in $(!/\text{W?}\text{-SC})$, β_1 must be cut-free,
- in $(\top\text{-CC})$, eA^\perp must be cut-free.

A reduction sequence is *standard* when made of standard contractions.

4.25. Theorem (Standardization Lemma).⁵ Let β be a proof-net and assume that there is a standard reduction from β to a cut-free β' . Then β is SN.

The delicate but boring proof is postponed. We immediately jump to the proof of the Strong Normalization Theorem.

4.26. Theorem (Strong Normalization Theorem). In PN2 all proof-nets are strongly normalizable.

Proof. The idea is to adapt the technique of ‘candidats de réductibilité’ of [1] to the case of PN2; in fact, the symmetry of linear negation makes it easier to some extent. During the proof, the Standardization Lemma (Theorem 4.25) is constantly used.

4.26.1. Definition. A *term of type A* is a proof-net β together with a distinguished conclusion which is an occurrence of A . In notations, we shall always make the harmless abuse that consists in considering the distinguished conclusion to be obvious from the context.

4.26.2. Definition. Let X be a set of terms of type A ; we define X^\perp as the set of those terms u of type A^\perp such that, for all t in X , the proof-net $\text{CUT}(t, u)$ (see Fig. 32) is SN

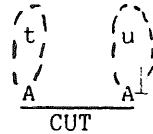


Fig. 32.

4.26.3. Proposition. (i) If $X \neq \emptyset$, then all elements of X^\perp are SN.

(ii) If all elements of X are SN, then X^\perp is nonvoid; it contains in fact the term $A \multimap A^\perp$ of type A^\perp .

⁵ The proof will be sketched in Section 6.

Proof. (i): If $\text{CUT}(t, u)$ is SN, then u is SN.

(ii): This proof already uses the Standardization Lemma: $\text{CUT}(t, u)$ contr t by a standard contraction when u is the axiom link. Hence, if u is SN, there is a standard reduction from u to a normal form, and therefore, from t : but then t is SN, too. \square

4.26.4. Definition. A CR (*candidat de réductibilité*) of type A is a set X of terms of type A such that

- (i) $X \neq \emptyset$;
- (ii) all terms of X are SN;
- (iii) $X = X^{\perp\perp}$.

4.26.5. Examples. A typical way to generate a CR of type A is to take the orthogonal Y^\perp of any nonvoid set Y of SN-terms of type A^\perp . In particular, taking Y to consist of an axiom link, we obtain the CR $\text{SN}(A)$ formed of all SN terms of type A .

4.26.6. Definition. Assume that $A[a]$ (where $a = a_1, \dots, a_n$ is a sequence of propositional variables) is a proposition, and let $B (= B_1, \dots, B_n)$ be a sequence of propositions; let $X (= X_1, \dots, X_n)$ be a sequence of CR's of respective types B . Then we define a CR $\text{RED}(A[X/a])$ of type $A[B/a]$, as follows:

- (i) $\text{RED}(\mathbf{1}[X/a]) = \{1\}^\perp$; $\text{RED}(\top[X/a]) = \text{SN}(\top)$;
- (ii) if A is a_i , then $\text{RED}(A[X/a])$ is X_i ;
- (iii) if A is $A' \& A''$, then $\text{RED}(A[X/a])$ is defined as the orthogonal of the set $\text{RED}(A'[X/a])^\perp \cdot \text{RED}(A''[X/a])^\perp$ formed of all proof-nets as shown in Fig. 33 with $t \in \text{RED}(A'[X/a])^\perp$, $u \in \text{RED}(A''[X/a])^\perp$.

$$\frac{\begin{array}{c} t \\ \text{---} \\ A'[B/a]^\perp & A''[B/a]^\perp \end{array}}{A[B/a]^\perp} \blacksquare$$

Fig. 33.

(iv) If S is $A' \& A''$, then $\text{RED}(A[X/a])$ is defined as the orthogonal of the union of the two sets:

$$\Downarrow^1(\text{RED}(A'[X/a]^\perp)) \quad \text{and} \quad \Downarrow^2(\text{RED}(A''[X/a]^\perp))$$

formed of all proof-nets obtained from a term t of $A'[X/a]^\perp$ (respectively $A''[X/a]^\perp$) by applying (1 \oplus) (respectively (2 \oplus)) (see Fig. 34).

$$\Downarrow^1_t \frac{\begin{array}{c} t \\ \text{---} \\ A'[B/a]^\perp \end{array}}{A[B/a]^\perp}^{1\oplus} \quad \Downarrow^2_u \frac{\begin{array}{c} u \\ \text{---} \\ A''[B/a]^\perp \end{array}}{A[B/a]^\perp}^{2\oplus}$$

Fig. 34.

(v) If A is $?A'$, then $\text{RED}(A[X/a])$ is defined as the orthogonal of the set $\$ \text{RED}(A'[X/a]^\perp)$ formed of all terms of type $!A'[B/a]^\perp$ of the form shown in Fig. 35.

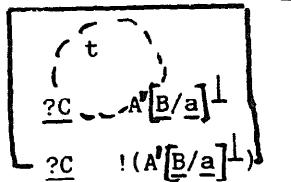


Fig. 35.

(vi) If A is $\wedge b.A'$, then $\text{RED}(A[X/a])$ is defined as the orthogonal of the set Z formed of all terms of type $\wedge b.A'^\perp[B/a]$ of the form shown in Fig. 36 such that, for some CR Y of type C , $t \in \text{RED}(A'^\perp[X, Y/a, b])$.

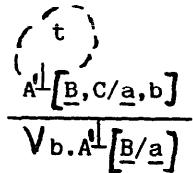


Fig. 36.

(vii) All other cases are treated by orthogonality, e.g., $\text{RED}(!A[X/a]) = \text{RED}(\wedge A^\perp[X/a]^\perp)$.

4.26.7. Lemma (Substitution Lemma)

$$\text{RED}(A[C/b][X/a]) = \text{RED}(A[X, \text{RED}(C[X/a])/a, b]).$$

Proof. This lemma simply states that the definition of reducibility is compatible with substitution. This is established without difficulty once the formalism has been understood. Remark that there is here a hidden use of second-order comprehension (to define a set by $\text{RED}(C[X/a])$). The proof of the Strong Normalization Theorem is not formalizable in PA_2 because of the use of the Substitution Lemma with unbounded C 's. This must be familiar to people who know the original proof for F . \square

4.26.8. Definition. A proof-net β with conclusions C is said to be *reducible* when the following holds: Let a be the list of all free variables of C and let us choose a sequence of formulas B and a sequence X of CR of types B (same length as a); select terms t in the CR $C^\perp[X/a]$ and make several cuts with all formulas of $C[B/a]$ as shown in Fig. 37. The result of these operations, called $\text{CUT}(\beta[B/a]; t)$, is SN.

4.26.9. Theorem. All proof-nets of PN2 are reducible.

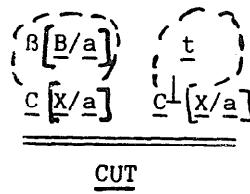


Fig. 37.

Proof. We argue by induction on a proof-net β . Since the substitutions B/a and X/a play no active role but make everything hard to read (and to write!), we shall not indicate them, working therefore with $\text{RED}(C^\perp)$, etc. The reader will be able to make the straightforward reconstruction himself.

Case 1: β is $A - A^\perp$; we have to show that, for any t, u in $\text{RED}(A^\perp), \text{RED}(A)$, $\text{CUT}(\beta; t, u)$ is SN. But there is a standard reduction from $\text{CUT}(\beta; t, u)$ to $\text{CUT}(t, u)$ which is SN by the definition of orthogonality; by standardization, $\text{CUT}(\beta; t, u)$ is SN.

Case 2: β is 1 ; we have to show that $\text{CUT}(\beta; t) \in \text{SN}$ for all $t \in \text{RED}(\perp)$. However $\text{RED}(\perp) = \{1\}^{\perp\perp}$; hence, $\text{CUT}(\beta; t) = \text{CUT}(\beta, t)$ is SN.

Case 3: β is $\overline{T - C}$; choose $t \in \text{RED}(0)$, $u \in \text{RED}(C^\perp)$; here a *general remark* must be made: Since $\text{RED}(0)^\perp = Z^\perp$, where Z consists of the axiom-link $\overline{T - 0}$, it is possible to replace $\text{RED}(0)$ by $Z!$ Hence, t is the axiom-link. Now it is very easy, using the fact that u are SN, to produce a standard normalization for $\text{CUT}(\beta; t, u)$.

Case 4: β comes from β' by the \perp -box in Fig. 38. Choose $t \in \text{RED}(1)$, choose $u \in \text{RED}(A^\perp)$ and consider $\text{CUT}(\beta; t, u)$. As we did in Case 3, one can restrict to $t = 1$, and there is a standard reduction from $\text{CUT}(\beta; t, u)$ to $\text{CUT}(\beta'; u)$ which is SN by hypothesis.



Fig. 38.

Case 5: β comes from β' by (\otimes) (see Fig. 39). After simplification (this refers to the remark made in Case 3), we are led to form $\text{CUT}(\beta; t \otimes u, v)$ with $t \in \text{RED}(A^\perp)$, $u \in \text{RED}(B^\perp)$, $v \in \text{RED}(C^\perp)$. $t \otimes u$ indicates the term obtained from t and u by (\otimes) .

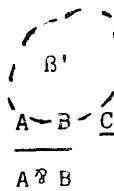


Fig. 39.

Now observe that there is a standard reduction sequence from $\text{CUT}(\beta; t \otimes u, v)$ to $\text{CUT}(\beta'; t, u, v)$, which is SN by the induction hypothesis.

Case 6: β is obtained from β' and β'' by (\otimes) (see Fig. 40). The induction hypothesis yields that $\text{CUT}(\beta'; t', u')$ is SN for all $t' \in \text{RED}(A')^\perp$ and $u' \in \text{RED}(C')^\perp$, which means that $\text{CUT}(\beta'; u') \in \text{RED}(A')$. For symmetric reasons, $\text{CUT}(\beta''; u'') \in \text{RED}(A'')$.

$$\frac{\begin{array}{c} \beta' \\ \beta'' \\ \hline C' \multimap A' \quad A'' \multimap C'' \end{array}}{A' \otimes A''}$$

Fig. 40.

$\text{RED}(A'')$. By tensorization,

$$\text{CUT}(\beta; u', u'') \in \text{RED}(A').\text{RED}(A'') \subset \text{RED}(A' \otimes A'')$$

by biorthogonality. But then $\text{CUT}(\beta; u', u'', v)$ is SN for any $v \in \text{RED}(A' \otimes A'')^\perp$.

Case 7: β is obtained from β' by $(1\oplus)$ (see Fig. 41). The induction hypothesis yields $\text{CUT}(\beta'; c) \in \text{RED}(A)$ for any $c \in \text{RED}(C)^\perp$; then $\text{CUT}(\beta'; c) \in \perp^1 \text{RED}(A) \subset \text{RED}(A \oplus B)$ and we are done.

$$\frac{\begin{array}{c} \beta' \\ \hline C \multimap A \end{array}}{A \oplus B} \perp \oplus$$

Fig. 41.

Case 8: β is obtained from β' by $(2\oplus)$: As Case 7.

Case 9: β is obtained from β' and β'' by the &-box in Fig. 42. After simplification, we see that we have to check if $\text{CUT}(\beta; c, \perp^1 t)$ and $\text{CUT}(\beta; c, \perp^2 u)$ is SN for any $c \in \text{RED}(C)^\perp$, $t \in \text{RED}(A)^\perp$, and $u \in \text{RED}(B)^\perp$. Now, the induction hypothesis yields that β'' is SN, and it is therefore possible to write a standard normalization from $\text{CUT}(\beta; c, \perp^1 t)$ to $\text{CUT}(\beta'; c, t)$, which is SN by induction hypothesis. For symmetric reasons, $\text{CUT}(\beta''; c, \perp^2 u)$ is SN.

$$\boxed{\frac{\begin{array}{c} \beta' \\ \beta'' \\ \hline C \multimap A \quad C \multimap B \\ C \multimap A \& B \end{array}}{A \& B}}$$

Fig. 42.

Case 10: β is obtained from β' by the !-box in Fig. 43. The induction hypothesis is that $\text{CUT}(\beta'; c, t)$ is SN for all $c \in \text{RED}(\mathbf{?}C)^\perp$ and t in $\text{RED}(A)^\perp$ and we want to conclude that $\text{CUT}(\beta; c, u)$ is SN for all u in $\text{RED}(\mathbf{!}A)^\perp$. Now, it is easy to show that we can make a simplification on the whole sequence c , namely that c is of the

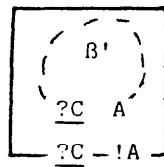


Fig. 43.

form $\mathbf{!d}$, for $d \in \text{RED}(C)^\perp$. But there is a standard reduction from $\text{CUT}(\beta; c, u)$ to $\text{CUT}(\mathbf{!CUT}(\beta'; c); u)$ (several (!-CC)). Now, the induction hypothesis says that $\text{CUT}(\beta'; c) \in \text{RED}(A)$; hence, $\mathbf{!CUT}(\beta'; c) \in \mathbf{!RED}(A) \subset \text{RED}(\mathbf{!A})$.

Case 11: β is obtained from β' by the weakening box in Fig. 44. We must show that $\text{CUT}(\beta; c, \mathbf{!t})$ is SN for all $c \in \text{RED}(C)^\perp$ and $t \in \text{RED}(A)^\perp$. Now t is SN, and there is a standard reduction from $\text{CUT}(\beta; c, \mathbf{!t})$ to $\text{CUT}(\beta'; c)$ plus weakenings, which is SN by induction hypothesis.

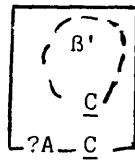


Fig. 44.

Case 12: β is obtained from β' by the dereliction rule shown in Fig. 45. We must show that $\text{CUT}(\beta; c, \mathbf{!t})$ is SN; but it reduces in a standard way to $\text{CUT}(\beta'; c, t)$, which is SN by induction hypothesis.

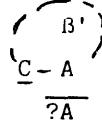


Fig. 45.

Case 13: β is obtained from β' by the contraction rule in Fig. 46. We must show that $\text{CUT}(\beta; c, \mathbf{!t})$ is SN; but it reduces in a standard way to $\text{CUT}(\beta; c, \mathbf{!t}, \mathbf{!t})$ plus contractions, which is SN by induction hypothesis.

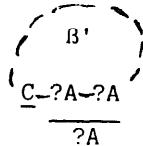


Fig. 46.

Case 14: β is obtained from β' by the \wedge -rule shown in Fig. 47. For the first time, we actually use the substitutions: we have to show that $\text{CUT}(\beta; c, t)$ is SN for any $c \in \text{RED}(C)^\perp$ and t coming from some u of some type $A^\perp[B/a]$ by the \vee -rule, s.t.

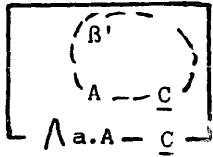


Fig. 47.

$u \in \text{RED}(A[X/a])^\perp$ for some CR X of type B . Now the induction hypothesis yields that $\text{CUT}(\beta'[B/a]; c, u)$ is SN, and observe the existence of a standard reduction from $\text{CUT}(\beta; c, t)$ to this proof-net.

Case 15: β is obtained from β' by the \vee -rule in Fig. 48. The induction hypothesis says that $\text{CUT}(\beta'; c) \in \text{RED}(A[B/a])$ for any c in $\text{RED}(C)^\perp$. Now, the Substitution Lemma says that $\text{RED}(A[B/a]) = \text{RED}(A[\text{RED}(B)/a])$; this shows that $\text{CUT}(\beta'; c) \in Z$ where Z is the set of existential terms such that $\text{RED}(\vee a.A) = Z^{\perp\perp}$; from $Z \subset Z^{\perp\perp}$, we conclude. \square

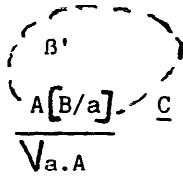


Fig. 48.

Proof of Theorem 4.26 (conclusion). Apply Theorem 4.26.9 to $B = X$ and $t = \text{axiom links}$. \square

5. Some useful translations

5.1. The translation of intuitionistic logic

The intuitionistic symbols $\wedge, \vee, \Rightarrow, \neg, \forall, \exists$ can be considered as defined in linear logic. The definitions are as follows:

$$\begin{array}{ll} A \wedge B = A \& B, & A \vee B = (!A) \oplus (!B), \\ A \Rightarrow B = (!A) \multimap B, & \neg A = (!A) \multimap 0, \\ \forall x A = \bigwedge x.A, & \exists x A = \bigvee x.!A. \end{array}$$

In particular, we see that we could have kept the symbols \wedge and \vee in linear logic, provided we have in mind *intuitionistic* conjunction and ‘for all’. The translation(.)⁰ from intuitionistic logic to linear logic is defined by means of the above dictionary, together with $A^0 = A$ when A is atomic.

The translation just given shows the unexpected heterogeneity of intuitionistic connectives (in sharp contrast with the translation into modal logic which is very regular, if nonconstructive). The most shocking point is the translation of negation:

one would have preferred \perp instead of $\mathbf{0}$, i.e., $(!A)^\perp$ for $\neg A$. However, intuitionistic negation has always been widely criticized. On the other hand Andrej Ščedrov pointed out to me the formal analogy between these translations and Kleene's 'slash', in particular where the places are determined where a “!” is needed. Even with this analogy in mind, it is hard to find a leading principle in this translation; for instance, the translation does not follow the principle of restricting to open facts, as a coarse analogy with the modal translation would suggest.

The translation $(.)^0$ is extended to a translation of proofs. The most efficient one is a translation of intuitionistic natural deduction (with $\neg A$ defined as $A \Rightarrow \mathbf{0}$) into linear sequent calculus. Here we use a version of linear sequent calculus with formulas on both sides of “ \vdash ”: $A \vdash B$ is short for $\vdash A^\perp, B$.⁶ The translation is defined by induction on the length of a deduction (d) of $(A)B$, i.e., of B under the hypotheses (A) : to such a (d) we associate $(d)^0$, a linear proof of $!A^0 \vdash B^0$.

(i) (d) is the deduction of $(B)B$ consisting of a hypothesis B ; then $(d)^0$ is

$$\frac{B^0 \vdash B^0}{!B^0 \vdash B^0} \ell D!$$

(ii) (d) is a deduction of $(A', A'')B' \wedge B''$ obtained from a deduction (d') of $(A')B'$ and a deduction (d'') of $(A'')B''$ by \wedge -introd; then $(d)^0$ is

$$\frac{\begin{array}{c} (d')^0 \\ !A'^0 \vdash B'^0 \end{array} \text{several } \ell W! \quad \begin{array}{c} (d'')^0 \\ !A''^0 \vdash B''^0 \end{array} \text{several } \ell W!}{!A'^0, !A''^0 \vdash B'^0 \& B''^0} \&.$$

(iii) (d) is a deduction of $(A)B'$ obtained from a deduction (d') of $(A)B' \wedge B''$ by the first \wedge -elim; then $(d)^0$ is

$$\frac{\begin{array}{c} (d')^0 \\ !A^0 \vdash B'^0 \& B''^0 \end{array} \quad \frac{B'^0 \vdash B'^0}{B'^0 \& B''^0 \vdash B'^0} \ell 1\&}{!A^0 \vdash B'^0} \text{CUT.}$$

(iv) (d) ends with the second \wedge -elim: symmetric to (iii).

(v) (d) is a deduction of $(A)B' \vee B''$, obtained from (d') of $(A)B'$ by the first \wedge -introd; then $(d)^0$ is

$$\frac{\begin{array}{c} (d')^0 \\ !A^0 \vdash B'^0 \\ !A^0 \vdash !B''^0 \end{array} \& !}{!A^0 \vdash !B'^0 \oplus !B''^0} \& 1 \oplus.$$

⁶ The rules are now written as left and right rules; e.g., the rule $(\&)$ now becomes $(\&)$ (on the right) and $(\ell \oplus)$ (on the left) etc.

(vi) (d) ends with the second \vee -introd: symmetric to (v).

(vii) (d) is a deduction of $(A, B', B'')D$ obtained from (e) of $(A)C' \vee C''$ and (d') of $(B', C')D$ and (d'') of $(B'', C'')D$ by \vee -elim, where C' and C'' are made of repetitions of C' and C'' respectively. Then $(d)^0$ is

$$\frac{\frac{\frac{\frac{!B'^0, !C'^0 \vdash D^0}{!B'^0, !C'^0 \vdash D^0} \text{ several } \ell C!, \text{ or one } \ell W!}{!B'^0, !C'^0 \vdash D^0} \text{ several } \ell W!}{!B'^0, !B''^0, !C'^0 \vdash D^0} \text{ (e)}^0}{!A^0 \vdash !C'^0 \oplus !C''^0} \quad \frac{\frac{!B''^0, !C''^0 \vdash D^0}{!B''^0, !C''^0 \vdash D^0} \text{ several } \ell C!, \text{ or one } \ell W!}{!B''^0, !C''^0 \vdash D^0} \text{ several } \ell W!}{!B''^0, !C''^0 \vdash D^0} \text{ (d'')}^0$$

$$\frac{!B'^0, !B''^0, !C'^0 \vdash D^0 \quad !B''^0, !C''^0 \vdash D^0}{!B'^0, !B''^0, !C'^0 \oplus !C''^0 \vdash D^0} \text{ CUT.}$$

$$\frac{!B'^0, !B''^0, !C'^0 \oplus !C''^0 \vdash D^0}{!A^0, !B'^0, !B''^0 \vdash D^0} \text{ CUT.}$$

(viii) (d) is a deduction of $(A)B \Rightarrow C$, obtained from a deduction (d') of $(A, B)C$ by \Rightarrow -introd, where B is made of repetitions of B ; then $(d)^0$ is

$$\frac{\frac{!A^0, !B^0 \vdash C^0}{!A^0, !B^0 \vdash C^0} \text{ several } \ell C! \text{ or one } \ell W!}{!A^0 \vdash !B^0 \multimap C^0} \text{ (d')}^0$$

(ix) (d) is a deduction of $(A', A'')B''$ obtained from (d') of $(A')B'$ and (d'') of $(A'')B' \Rightarrow B''$ by \Rightarrow -elim; then (d) is

$$\frac{\frac{\frac{!A'^0 \vdash B'^0}{!A'^0 \vdash B'^0} \text{ (d')}^0}{!A'^0 \vdash !B'^0 \multimap B''^0} \text{ (d'')}^0}{!A'^0, !B'^0 \multimap B''^0} \quad \frac{\frac{B''^0 \vdash B''^0}{!A'^0, !B'^0 \multimap B''^0 \vdash B''^0} \ell \multimap}{!A'^0, !B'^0 \multimap B''^0 \vdash B''^0} \text{ CUT.}$$

(x) (d) is a deduction of $(A)B$ coming from a deduction of $(A)\mathbf{0}$ by the rule $\mathbf{0}$ -elim (we have used the notation $\mathbf{0}$ for the absurdity of intuitionistic logic); then $(d)^0$ is

$$\frac{\frac{!A^0 \vdash \mathbf{0}}{!A^0 \vdash \mathbf{0}} \quad \frac{\mathbf{0} \vdash E^0 \text{ (axiom)}}{!A^0 \vdash E^0}}{!A^0 \vdash B^0} \text{ CUT.}$$

(xi) (d) is a deduction of $(A)\forall x.B$ coming from a deduction (d') of $(A)B$ by the rule \forall -introd; then $(d)^0$ is

$$\frac{\frac{!A^0 \vdash B^0}{!A^0 \vdash B^0} \text{ (d')}^0}{!A^0 \vdash \bigwedge x.B^0 \text{ } \multimap \bigwedge}.$$

(xii) (d) is a deduction of $(A)B[t/x]$ coming from a deduction (d') of $(A)\forall x.B$ by the rule \forall -elim; then $(d)^0$ is

$$\frac{\frac{\frac{(d')^0}{!A^0 \vdash \bigwedge x.B^0} \quad \frac{B^0[t/x] \vdash B^0[t/x]}{\bigwedge x.B^0 \vdash B^0[t/x]}}{\ell \wedge}}{\text{CUT.}}$$

(xiii) (d) is a deduction of $(A)\exists x.B$ obtained from a deduction (d') of $(A)B[t/x]$ by the rule \exists -introd; then $(d)^0$ is

$$\frac{(d')^0}{!A^0 \vdash B^0[t/x] \quad \text{i}\vee}.$$

(xiv) (d) is a deduction of $(A, B)D$ obtained from (e') of $(A)\exists x.C$ and (e'') of $(B, C)D$ by \exists -elim; then (d^0) is

$$\frac{\frac{\frac{(e'')^0}{!B^0, !C^0 \vdash D^0} \quad \frac{!B^0, !C^0 \vdash D^0}{\text{several } \ell C! \text{ or one } \ell W!}}{\ell \vee}}{\frac{!A \vdash \bigvee x!C^0 \quad \frac{!B^0, \bigvee x!C^0 \vdash D^0}{!B^0, !C^0 \vdash D^0}}{\text{CUT.}}}$$

Remark. If we start with a deduction which is normal (in the strongest sense, involving commutative conversions), then the construction of $(d)^0$ can be slightly modified, yielding a normal proof.

Observe that our translation is faithful in the sense that if A^0 is provable in linear logic, A is provable in intuitionistic logic. This can be easily justified as follows: Linear logic satisfies cut-elimination, hence a subformula property. A^0 is written in the fragment $\mathbf{0}, \neg, \oplus, \&, !, \wedge, \vee$ of linear logic. In particular, writing the rules for this fragment with two-sided sequents we see that the proof of A^0 uses only intuitionistic linear sequents. Now, if we erase all symbols $!$, and replace $\oplus, \&, \neg, \wedge, \vee$ by $\vee, \wedge, \Rightarrow, \forall, \exists$, then we get a proof of A in intuitionistic logic.

The translation just chosen has been historically spoken the first work on linear logic, dating back to the end of 1984 for disjunction, and October 1985 for the full language. The translation is sound, not only for provability, but also w.r.t. the coherent semantics. Once more, this is the coherent semantics of intuitionistic logic which suggested to put the hidden linear features on the front stage. Other translations are possible, for instance one following the idea of sticking to open facts:

$$A^* = !A \quad \text{for } A \text{ atomic,}$$

$$(A \wedge B)^* = A^* \otimes B^*, \quad (A \vee B)^* = A^* \oplus B^*,$$

$$(A \Rightarrow B)^* = !(A^* \multimap B^*), \quad \mathbf{0}^* = \mathbf{0},$$

$$(\forall x A)^* = !\wedge A^*, \quad (\exists x A)^* = \bigvee x A^*.$$

This boring translation is reminiscent of the modal translation of intuitionistic logic. This translation is not sound w.r.t. the coherent semantics and this is enough to show that its interest is limited.

5.2. The translation of the system F

F is nothing but a functional notation for second-order propositional natural deduction, so we can essentially apply the translation of Section 5.1. We concentrate here on the reduced version of F (variables, \Rightarrow , \forall), and we make a slightly more precise definition so that normal terms of F are directly translated as normal proof-nets.

5.2.1. Types

The types of F are translated as follows:

$$a^0 = a \text{ when } a \text{ is a variable,}$$

$$(S \Rightarrow T)^0 = !S^0 \multimap T^0, \quad (\forall a. T)^0 = \bigwedge a. T^0.$$

The translation has the *substitution property*: $(S[T/a])^0 = S^0[T^0/a]$.

5.2.2. Translation of terms

We distinguish two kinds of terms:

(i) General terms of $t[x]$: x is a sequence of variables of types S , which are exactly the free variables of t which is of type T . Such a term will be translated into a proof-net t^0 whose conclusions will be exactly $!S^{0\perp}$ and T^0 .

(ii) Terms with a distinguished headvariable y : $t[x, y]$ with x of type S , y of type H , t of type T . y is distinct from all variables in t . When we say that y is distinguished, this only means that we have decided to remark that we are in case (ii), instead of using the more general procedure case of (i). In that case, t^0 has the conclusions $?S^{0\perp}, H^{0\perp}, T^0$.

Step 1: $t = y$ (so that $T = H$); then t^0 is

$$\overline{T^{0\perp} \quad T^0!}$$

Step 2: t has a headvariable and may be written as $u[y(v)/y']$, where $u[y']$ is already a term with a headvariable. The other variables of u and v are x and x' , some of them being common. Then we form the configuration shown in Fig. 49 (y

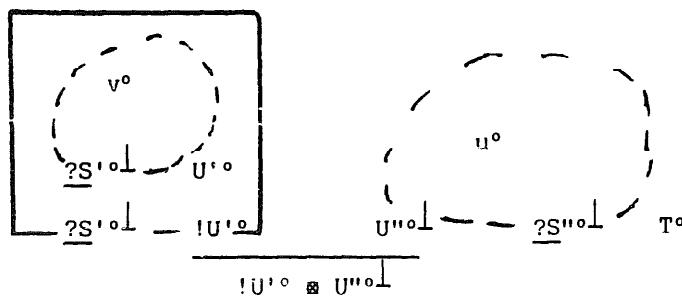


Fig. 49.

is of type $U' \Rightarrow U''$ so that we must get a conclusion $!U'^0 \otimes U''^0 \perp$). From this, by using the rule

$$\frac{?S_i'^0 \perp \quad ?S_j''^0 \perp}{?S_i'^0} ?C$$

every time $x'_i = x''_j$, we obtain the interpretation t^0 .

Step 3: t has a headvariable and may be written as $u[u\{U'\}/y']$ for a certain term $u[y']$ (which already has a headvariable y' of type $U''[U'/a]$, y being of type $\wedge a.U''$). We form the configuration in Fig. 50.

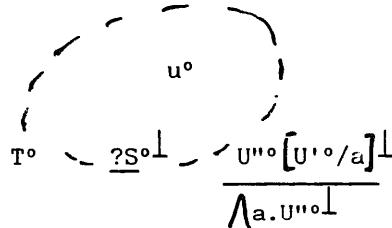


Fig. 50.

Step 4: t has a headvariable, but we do not want to distinguish it any longer. Consider the term $u[y, x]$ with a new distinguished headvariable such that t is $u[x_i, x]$; then form the configuration in Fig. 51. This will be t^0 , except if x_i is equal to some variable of x , in which case one must end with a contraction.

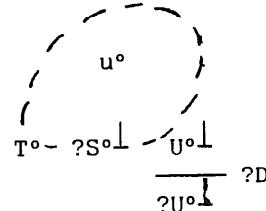


Fig. 51.

Note: All other steps are steps involving no terms with headvariables.

Step 5: t is $u(v)$; u is of type $T' \Rightarrow T''$, v is of type T' : we now form the configuration in Fig. 52 to which we apply, if necessary, a certain number of contractions between $?S_i'^0 \perp$ and $?S_j''^0 \perp$ when $x'_i = x''_j$.

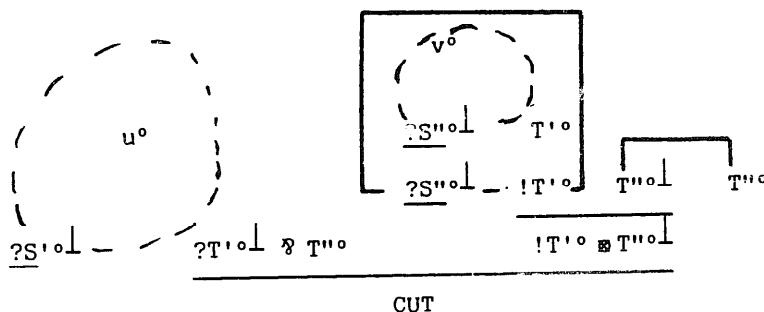


Fig. 52.

Step 6: t is $u\{U\}$ with u of type $\forall a.T'$; the corresponding configuration is shown in Fig. 53.

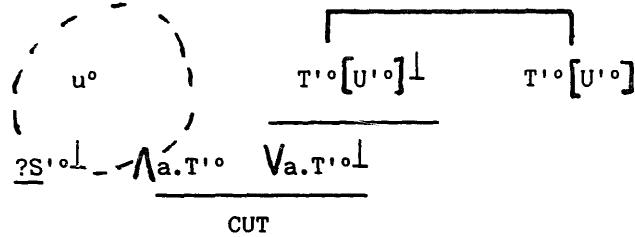


Fig. 53.

Step 7: t is $\lambda x.u$, with u, x of types T'', T' . Let u' be u^0 if x occurs in u and otherwise, let u' be the result of a $W?$ on $?T'^0 \perp$ applied to u^0 (see Fig. 54).

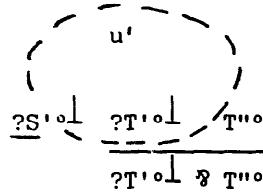


Fig. 54.

Step 8: t is $\forall a.u$, with u of type T' , the configuration is given in Fig. 55.

The steps must now be put together to form a correct definition by induction of the translation $()^0$. The details are left to the reader, but, by correctly handling the case with headvariables, one translates a normal term of F into a cut-free proof-net of PN2.

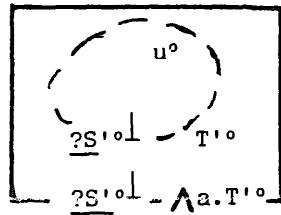


Fig. 55.

5.2.3. Semantic soundness of the translation

The coherent semantics of F has been given in [5]; we also have a coherent semantics for PN2 (Section 3); the semantic soundness of the translation is that $t^{0*} = t^*$ if the symbol $*$ denotes both semantic interpretations.

(1) The first thing to check is that there is a semantic soundness of the translation of types, namely that $T^{0*} = T^*$. This easily follows from two remarks:

- (i) *the type constructor \forall is exactly the same thing as \wedge .*
- (ii) *semantically speaking, $X \Rightarrow Y = !X \multimap Y$.*

Proof: $X \Rightarrow Y$ is the set of traces of stable functions from X to Y . This coherent space is defined by

$$|X \Rightarrow Y| = X_{\text{fin}} \times |Y|$$

where X_{fin} is the set of all finite objects of X .

$$(a, z) \subset (b, t) [\text{mod } X \Rightarrow Y] \text{ iff (1) } a \cup b \in X \rightarrow z \subset t [\text{mod } Y],$$

$$\text{(2) } a \cup b \in X \text{ and } a \neq b \rightarrow z \neq t.$$

Now, observe $!X$; we have $!|X| = X_{\text{fin}}$; moreover, $a \subset b$ [mod $!X$] iff $a \cup b \in X$; in other terms, $|X \Rightarrow Y| = |!X| \times |Y|$ and

$$(a, z) \subset (b, t) [\text{mod } X \Rightarrow Y] \text{ iff (1) } a \subset b [\text{mod } !x] \rightarrow z \subset t [\text{mod } Y],$$

$$\text{(2) } a \cap b [\text{mod } !X] \rightarrow z \cap t [\text{mod } Y].$$

This clearly established that $X \Rightarrow Y = !X \multimap Y$ (*end of proof*).

(2) The semantic soundness of the interpretation of term is based on the ideas already introduced in the *pons asinorum* (Section IV); checking all the cases here would be a pure loss of time.

5.2.4. Syntactic soundness of the translation

This is another issue, albeit of slightly less importance than the semantic soundness. The semantic soundness is enough to ensure that there is no loss of expressive power between F and PN2. (There is no essential gain either: the Normalization Theorem for PN2 can be carried out in PA_2). However, a syntactic relation between normalization in F and PN2 would be welcome. In fact, when $t = / u$ in F , one can find a u' such that $t^0 = / u'$ in PN^2 and u' differs from u^0 by a different order of use of the rules ($C?$). This problem is due to the fact that, in F (and systems based on λ -abstraction), the identification of variables is indeed made just before the abstraction, while PN2 is more refined and gives a specific order for the identification. This is a quality of PN2: a proof-net of PN2 contains additional information which the theory usually ignores (the order of the contractions), but which is very relevant in practice (e.g., for retaining the substitution as long as possible) and so traditionally belongs to the sphere of ‘bricolage’ (i.e., handicraft).

The verification of the syntactic soundness of the translation is boring and without surprise.

5.3. Translation of current data in PN2

Since we know that current data types can be translated in F and that F translates in PN2, we can, by transitivity, translate integers, booleans, trees, lists, etc. into PN2. However, PN2 has subtler handling of the types and we often find a better translation, not transiting through F as will be shown in the following subsections.

5.3.1. Booleans

The best is to choose the definition $\text{bool} = 1 \oplus 1$, with

$$\text{TRUE} = \frac{1}{\text{bool}} 1 \oplus, \quad \text{FALSE} = \frac{1}{\text{bool}} 2 \oplus.$$

The instruction **IF** β **THEN** β' **ELSE** β'' is defined by the configuration in Fig. 56, where β and β' have the same conclusions C . It is immediate that **IF** **TRUE** **THEN** β' **ELSE** $\beta'' = / \beta'$ and **IF** **FALSE** **THEN** β' **ELSE** $\beta'' = / \beta''$. There are however two strategies possible for normalizing such instructions.

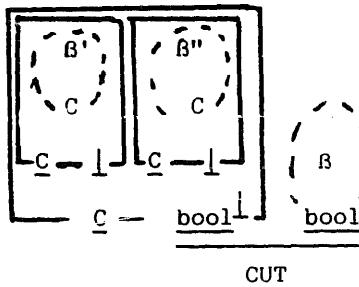


Fig. 56.

(i) The *lazy strategy* consists in not using the commutative conversions (&-CC). The **IF-THEN-ELSE** instruction is viewed as a sealed box from the viewpoint of β' and β'' . The only way to get the box opened is to wait until the main door is opened (cut on **TRUE** or **FALSE**). This strategy is particularly interesting in the case we are normalizing a procf-net whose conclusions involve neither \oplus nor \vee since we are sure that all such boxes will eventually be opened by the main door. In terms of parallelism, such a box can be viewed as a moment when one has to wait (the communication through C is temporarily interrupted) (and maybe this waiting time can be used to do other tasks).

(ii) The *general strategy* consists in entering the boxes even by the auxiliary doors; this causes a duplication of certain tasks, with the unpleasant feature that if the main door is eventually opened, half of what has been done may be erased. But if we have indications that the main door may never open, this strategy may be of some use.

5.3.2. Integers

In F integers are translated as $\forall a. a \Rightarrow (a \Rightarrow a) \Rightarrow a$. The straightforward translation of F into PN2 would yield three “!”s, including a nested one! It turns out that, among the three implications used in this type, the first two can be taken as linear. We therefore define **int** as $\wedge a. a \multimap ((a \multimap a) \Rightarrow a)$. Equivalently, **int** = $\wedge a. a \multimap (! (a \multimap a) \multimap a)$, and this is isomorphic to $\wedge a. !(a \multimap a) \multimap (a \multimap a)$. For instance, the number “3” can be represented in F (natural deduction) as shown in Fig. 57(a). In PN2, the representation of “3” looks as shown in Fig. 57(b).

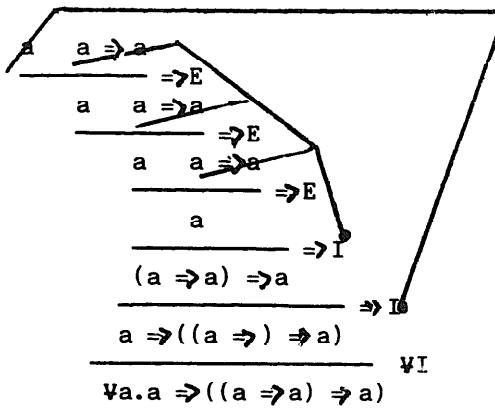
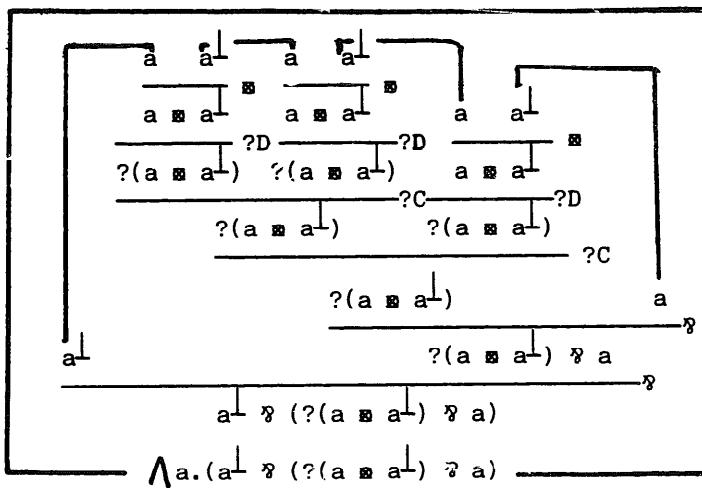
Fig. 57(a). The number “3” represented in the system F (natural deduction).

Fig. 57(b). The number “3” represented in PN2.

The functional notation for “3” in F is $\forall a.\lambda x^a.\lambda y^{a \Rightarrow a}.y(y(a))$. There are other possible representations of “3” in PN2: before the first \wp -rule, we can use a different combination of the rules ($C?$) and maybe some rules ($W?$).

Consider all possible proof-nets with int as conclusion; the end is necessarily as in the described example a \wedge -box, then a \wp -link whose premises are a^\perp and $?(\alpha \otimes a^\perp) \wp a$, which in turn follows from a \wp -link applied to a and $?(\alpha \otimes a^\perp)$. Now, there are several possibilities: this $?(\alpha \otimes a^\perp)$ may arise by several combinations of ($W?$), ($C?$), and ($D?$). Above the combination of these rules there are several premises $a \otimes a^\perp$, and above them premises a, a^\perp . These premises are linked together by axiom-links, and are also linked to the formulas a and a^\perp mentioned much earlier. The only possible configuration, in terms of proof-nets is the one shown in Fig. 58 and the number of axiom-links determines which integer we are speaking of: $n + 1$ axiom-links are needed to represent n . For instance, Fig. 57(b) was using four axiom-links, and therefore represents “3”. But any other variation on this picture (with extra ($W?$), different order for the rules ($C?$)) would yield a term with the same semantics.

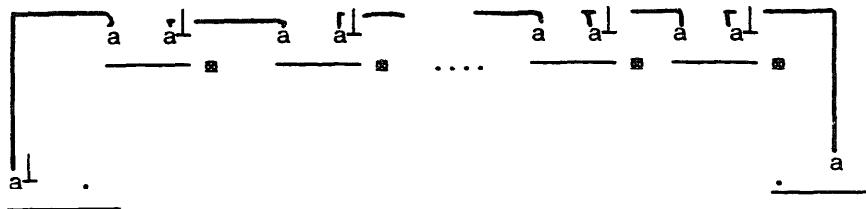


Fig. 58.

Let us explain the interpretation of the proof-net "3"; for this, the best is to return to the *pons asinorum* (Section IV). Specify a coherent space X in order to interpret a in the inner part of the box. Now, $X^\perp \wp (?(\mathcal{X} \otimes X^\perp) \wp X)$ means $X \multimap (!X \multimap X) \multimap X$, and should be viewed as the set of all linear maps from X to $!(X \multimap X) \multimap X$. Then, $!(X \multimap X) \multimap X$ is $(X \multimap X) \Rightarrow X$ and should be seen as the set of all stable maps from $X \multimap X$ to X . Summing up, what is inside the box describes, for $a = X$, a binary stable function mapping $X, X \multimap X$ into X , and linear in the first argument. This function is defined by $F(x, f) = f(f(f(x)))$, as the semantic computation easily shows. More generally, all proof-nets which represent n are semantically identical: on X , they correspond to the function $F(x, f) = f^n(x)$.

This is very close to the more familiar representation of integers in F ; the main difference is that the argument f is linear. It is possible to stick to 'f linear' because f^n is still linear. But the function associating f^n with f is never linear, but for $n = 1$. Now, look at Fig. 59; the diagram represents the successor function, which linearly

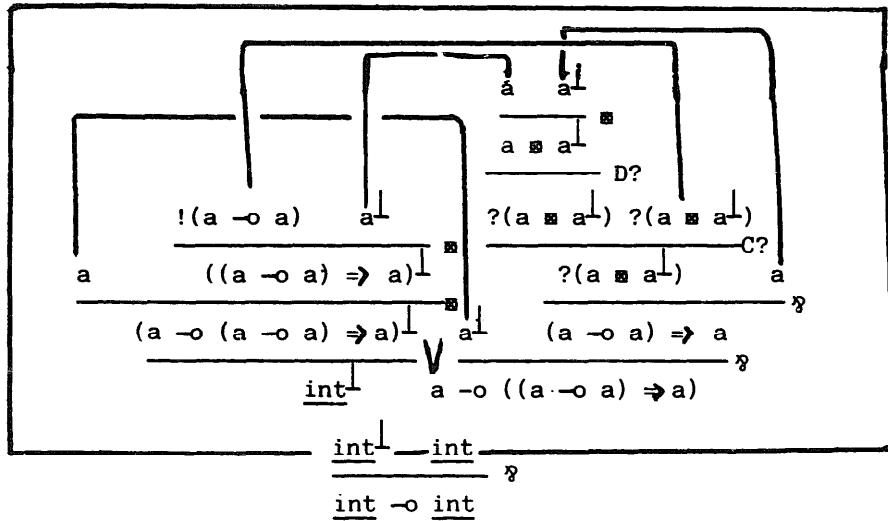


Fig. 59.

maps int into int ; if one applies $SUC\cup$ to a representation \bar{n} of the integer n , by means of the configuration in Fig. 60, then, after normalization, one gets a representation $\bar{n+1}$ of the next integer.

As usual, int allows definitions by primitive recursion: typically, from a proof-net β with A as conclusion and from β' with $A \multimap A$ as conclusion, one can form

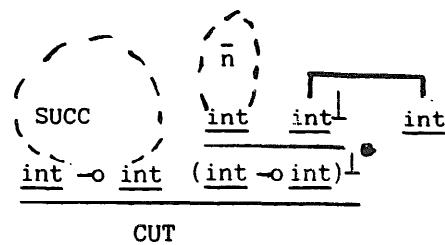


Fig. 60.

$\text{REC}(\beta, \beta')$ with the conclusion $\text{int} \multimap A$ (see Fig. 61). (In fact, there is no need that A and $A \multimap A$ should be the unique conclusions of β or β' ; arbitrary other conclusions are welcome in β and additional ?-conclusions can be accepted in β' .)

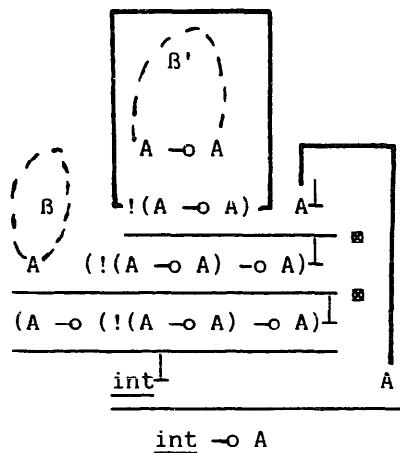


Fig. 61.

When we apply $\text{REC}(\beta, \beta')$ to \bar{n} , by the configuration in Fig. 62(a), this reduces to the configuration in Fig. 62(b) (n occurrences of β' ; this proof-net is “ β' applied n times to β ”).

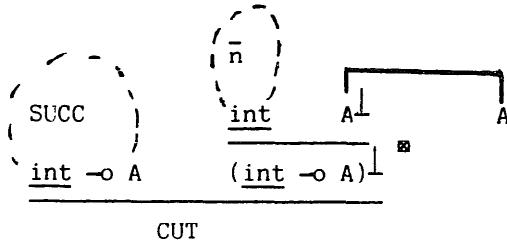
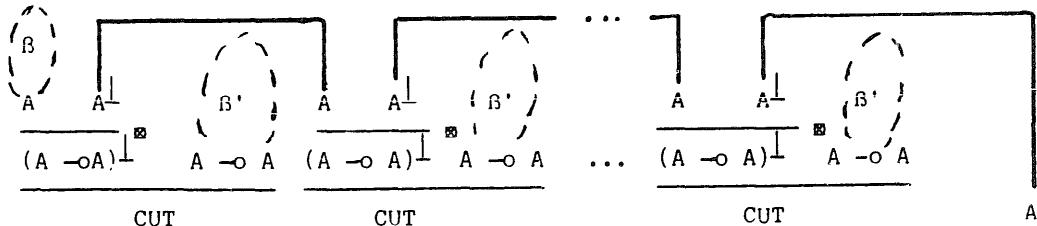


Fig. 62(a).

Fig. 62(b). “ β' applied n times to β ”.

The problem is that, in practice, we do not necessarily have linear functions to start with when making recursion. However, what about a recursion with β as a proof of A , β' as a proof of $A \Rightarrow A$, i.e., $!A \multimap A$? This can be reduced to the previous case: from the configuration in Fig. 63 (here, auxiliary conclusions can be accepted

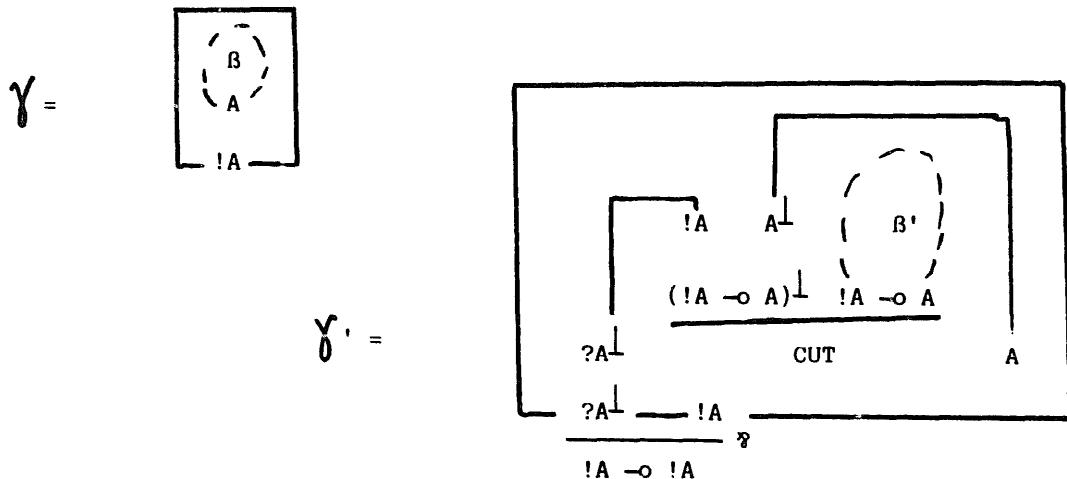


Fig. 63.

too, but only of the ?-form). Then, what we want can be expressed by derelicting $\text{REC}(\gamma, \gamma')$ as shown in Fig. 64.

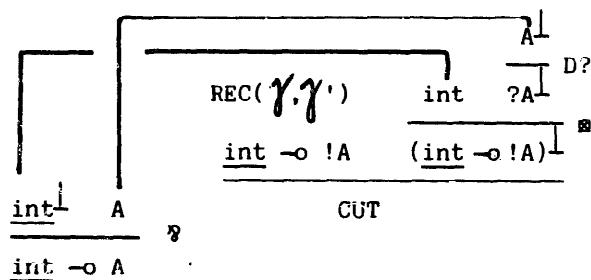


Fig. 64.

Exercise. (i) Show the existence of a proof-net whose conclusion is $\text{int} \multimap \text{!int}$ and such that when applied to an integer \bar{n} , it yields $!\bar{n}$; i.e., it is put into a !-box.

(ii) Conclude that the same functions from \mathbb{N} to \mathbb{N} are representable in PN2 as of type $\text{int} \multimap \text{int}$ and of type $\text{int} \Rightarrow \text{int}$.

5.3.3. Lists, trees

We can take more or less the usual translations while being careful to use linear implications everytime \Rightarrow is not actually needed. The details would be too much here.

5.4. Interpretation of classical logic

Of course, there is the $\neg\neg$ -translation of Gödel, which reduces the problem to the one already solved in Section 5.1. However, the $\neg\neg$ -translation has a terrible

defect, namely using intuitionistic negation which is the only citicizable intuitionistic connective. But Gödel's translation is still working if, instead of $\neg\neg A$, one uses $((A \Rightarrow F) \Rightarrow F)$ where F is any formula which is not provable. Here we shall use \perp for F . Formulas of the form $((A \Rightarrow \perp) \Rightarrow \perp)$ are equivalent to $?!A$; i.e., in toplinear terms, they are closed facts, equal to the closure of their interior. Let us call a fact *regular* when it is equal to the closure of its interior, i.e., to the closure of some closed fact. We have the following properties.

- (i) If A is regular, then $?(A^+)$ is regular.
- (ii) If A and B are regular, then $A \wp B$ is regular: $A \wp B = ?!A \wp ?!B = ?(!A \oplus !B)$ where $?!A \oplus ?!B$ is open.

From this, the principles of the translation are immediate:

$$\begin{aligned} A^+ &= ?!A \quad \text{when } A \text{ is atomic} \\ (A \vee B)^+ &= A^+ \wp B^+, \quad (\neg A)^+ = ?(A^{+\perp}), \quad (\forall x A)^\perp = ?! \wedge x. A^+. \end{aligned}$$

The interpretation of classical disjunction is particularly simple. This justifies our claim that *par is the constructive contents of classical disjunction*.

Now, the regular subsets of a toplinear space form a complete boolean algebra and from this, it will be possible to justify all classical laws translated by $()^+$. In particular, to some extent, we get a semantics of proofs for classical logic. But the translation used lacks the purity obtained in the intuitionistic case. This is due to the fact that the equivalence $?!A^+ \circ\circ A^+$ is used too often.

5.5. Translation of cut-free classical logic

If we turn our attention towards cut-free classical logic, the situation changes radically, in the sense that a convincing interpretation inside linear logic is possible. The interpretation is based on the familiar three-valued idea of giving independent meanings to positive and negative occurrences of formulas (see [3]).

By induction on the formula A , we define pA and nA :

$$\begin{aligned} pA &= nA = A \quad \text{when } A \text{ is atomic} \\ p\neg A &= (nA)^\perp, & n\neg A &= (pA)^\perp, \\ pA \vee B &= (pA) \oplus (pB), & nA \vee B &= !(nA) \wp !(nB), \\ pA \wedge B &= ?(pA) \otimes ?(pB), & nA \wedge B &= (nA) \& (nB), \\ pA \Rightarrow B &= (nA)^\perp \oplus (pB), & nA \Rightarrow B &= ?(pA) \multimap !(nB), \\ p\forall x A &= \wedge x. ?(pA), & n\forall x A &= \wedge x. nA, \\ p\exists x A &= \vee x. pA, & n\exists x A &= \vee x. !nA. \end{aligned}$$

The reader will toy a little with the definition to see that this definition is built on strong symmetries, unlike the less satisfactory translation in Section 5.4. In fact,

the situation is the same between the very good translation ()⁰ and the less satisfactory translation ()*, both considered in Section 5.1. But here we have to give up the cut-rule.

Now, with each proof in cut-free classical logic of a sequent $A \vdash B$, one associates a proof of the sequent $!nA \vdash ?pB$ in linear logic in a more or less straightforward way. This paper has already been too long, and a reader not already dead at this point will have no trouble in finding out the details. In particular, due the cut-elimination theorem, *there is a very nice and natural semantics of proofs of full classical logic*.

The cut-rule has a very ambiguous status: it can be seen as the general scheme: $?pA \multimap !nA$, i.e., $n(A \Rightarrow A)$. This scheme, although wrong, is conservative over sequents of the form $!nB \vdash ?pC$, i.e., over formulas of the form $?pA$, whose main feature seems to be the absence of “!”.

Conjecture and question. It should be possible to find a formulation of cut-elimination as a general conservativity result over a fragment of linear logic, e.g., the fragment free from $!$. The cut-elimination procedure would appear as the effective way of eliminating the conservative principles (which should be the formulas $n(A \Rightarrow A)$ or something more general). *Such a program, if properly carried out, would give a full constructive content to classical logic.*

5.6. The Approximation Theorem

In the introduction we have already mentioned that linear logic was able to control the length of disjunctions in Herbrand's theorem. Let us now use what we know, in particular, the translation of classical logic just given. To simplify the matter, we shall work with a prenex formula $\exists x \forall y \exists z \forall t Rxyzt$ with R quantifier-free. The p -translation of such a formula is $\forall x \wedge y ? \forall z \wedge t ?Sxyzt$ with S quantifier-free and $!$ -free.

Now, if our formula is provable in classical logic, the $?$ of its p -translation, i.e., $? \forall x \wedge y ? \forall z \wedge t ?Sxyzt$ will be provable in linear logic.

Now, consider the connectives $?_n$ defined by

$$?_n A = (\perp \oplus A) \wp (\perp \oplus A) \wp \cdots \wp (\perp \oplus A) \quad (n \text{ times}).$$

The Approximation Theorem (see below) says that we can replace each $?$ by approximants $?_n$; in particular, $?_n \forall x \wedge y ?_m \forall z \wedge t ?Sxyzt$ (in the quantifier-free part, we have kept $?$). This formula clearly expresses that we had a midsequent with at most $n \cdot m$ formulas or, equivalently, a Herbrand disjunction of length at most $n \cdot m$. It also says something about the intimate structure of this Herbrand disjunction (the n and the m).

This example clearly shows how linear logic can be used to control the length (and also) the structure of Herbrand disjunctions. Now, the approximation theorem we were speaking about is just the mathematical contents of our slogan: *usual logic is obtained from linear logic (without modalities) by a passage to limit.*

5.1. Definition (approximants). The connectives $!$ and $?$ are approximated by the connectives $!_n$ and $?_n$ ($n \neq 0$):

$$!_n A = (\mathbf{1} \& A) \otimes \cdots \otimes (\mathbf{1} \& A) \quad (n \text{ times}),$$

$$?_n A = (\perp \oplus A) \wp \cdots \wp (\perp \oplus A) \quad (n \text{ times}).$$

5.2. Theorem (Approximation Theorem). Let A be a theorem of linear logic; with each occurrence of $!$ in A , assign an integer $\neq 0$; then it is possible to assign integers $\neq 0$ to all occurrences of $?$ in such a way that if B denotes the result of replacing each occurrence of $!$ (respectively $?$) by $!_n$ (respectively $?_n$) where n is the integer assigned to it, then B is still a theorem of linear logic.

Proof. One can formulate the exact analogue for a sequent $\vdash A$ and we prove the result by induction on a cut-free proof of $\vdash A$ in linear sequent calculus. We content ourselves with a few interesting cases:

(i) One can assume that the axioms $\vdash A, A^\perp$ are restricted to the atomic case and so, no problem.

(ii) If the last rule is $(!)$: from $\vdash A, ?B$ infer $\vdash !A, ?B$, then we already obtained $\vdash A', ?_n B'$ (approximation A' of A , B' of B , and a sequence n). Now, one easily gets $\vdash \mathbf{1}, ?_n B'$ and so, $\vdash \mathbf{1} \& A', ?_n B'$. Let k be the integer associated with the first $!$ in $!A$; we get $\vdash !_k A', ?_{nk} B'$.

(iii) If the last rule is $(D?)$: from $\vdash ?A, ?A, B$ infer $\vdash ?A, B$ and if we have already approximations $\vdash ?_n A', ?_m A'', B$, we can first ensure $A' = A''$ by increasing the respective $?$ -assignments. Then let A''' be the result; we then form $\vdash ?_{n+m} A''', B'$.

(iv) If the last rule is $(\&)$: from $\vdash A, C$ and $\vdash B, C$ infer $\vdash A \& B, C$, then we have approximants $\vdash A', C'$ and $\vdash B', C''$; by increasing the $?$ -assignments in C' and C'' , we get C''' and $\vdash A' \& B', C'''$ is still provable.

(v) We have gone through the main steps: The rule $(D?)$ would involve a $?_1$ and the rule $(W?)$ a $?_0$ (which we have excluded, so a $?_1$, too).

The crucial fact that one can always replace $?_n$ by $?_m$ when $n < m$ is more or less immediate. In a similar way, $!_m$ can be replaced by $!_n$ when $n < m$. \square

6. Work in progress: slices

There are several directions of work in progress, e.g., the theory of totality (solved by saying that an element of a coherent space X is total with the phase p), which have not reached maturity and which have therefore been omitted from this paper, which is already a bit long. However, the presentation of *slices* has been finally decided, for two reasons:

(i) this is the only immediate approach we know to the boring standardization theorem;

(ii) it presents the absolute limit for a parallelization of the syntax, i.e., the removal of all boxes but !-ones.

The present stage of the theory is a list of definitions with no theorem....

6.1. Definition. A *slice* is a proof-structure making use of (1) all links and axioms already introduced, (2) !-boxes (the contents of such boxes are sets of slices all ending with $?B, A$), and (3) unary rules:

$$\frac{A}{\Lambda a.A} \wedge, \quad \frac{A}{A \& B} 1\&, \quad \frac{B}{A \& B} 2\&.$$

6.2. Definition. The *slicing* of a proof-net β is a nonempty set of slices with the same conclusions as β .

(1) *Slicing of a box*: the !-box built from β slices into the set consisting of just one !-box; the contents of this box is the slicing of β . Boxes as shown in Fig. 65(a) slice into the disconnected structures of Fig. 65(b), where the β_i 's are the slices of

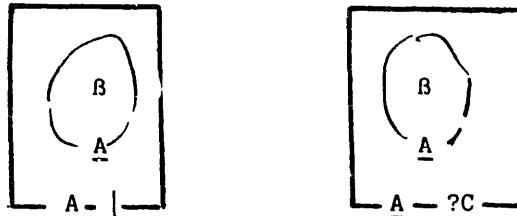


Fig. 65(a).



Fig. 65(b).

β . A box of the type shown in Fig. 66(a) slices into the structures of Fig. 66(b), where the β_i 's and β'_j 's are the respective slices of β and β' . Finally, a box as in

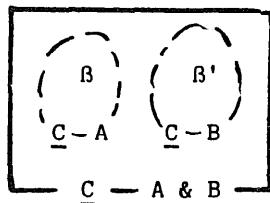


Fig. 66(a).

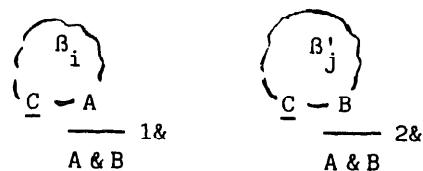


Fig. 66(b).

Fig. 67(a) slices into the structures of Fig. 67(b) where the β_i 's are the slices of β .

An axiom $\Box T - C$ (seen as a box) slices into the disconnected structure $T \ C$.

(2) If β is a proof-net built from boxes B_1, \dots, B_n by means of links, say $\beta(B_1, \dots, B_n)$, then the slices of β are all $\beta(\beta_1^i, \dots, \beta_n^i)$ for all slices $\beta_1^i, \dots, \beta_n^i$ of B_1, \dots, B_n respectively.

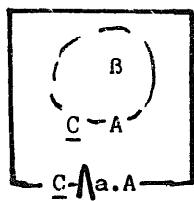


Fig. 67(a).

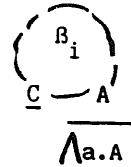


Fig. 67(b).

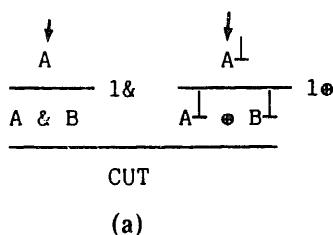
6.3. Remarks. (i) The slicing is the ‘development’ of a proof-net by ‘distributivity’. Each slice is in itself logically incorrect, but it is expected that the total family of slices has a logical meaning. However, there is no characterization of sets of slices which are the slices of a proof-net. Moreover, we do not even know whether or not identifying proof-nets with the same slices could lead to logical atrocities; for instance, is it possible to define the coherent semantics of a set of slices?

(ii) The slicing does not work for !-boxes; this is because the modalities ! and ? are the only nonlinear operations of linear logic. However, if we were working with ! as an infinite tensorization $\otimes 1 \& A$, then it would be possible to slice, but we would get a nondenumerable family of slices! A more reasonable approach would be to make finite developments based on $!A = (1 \& A) \otimes !A$ so that we never go to the ultimate, nondenumerable slicing, but generate it continuously. This idea is of interest because it could serve, without changing anything essential to PN2, for expressing nonterminating processes.

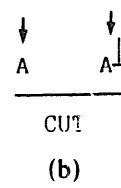
(iii) We shall now try to define the normalization procedure on the slices directly; however, the rule (T-CC), which involves the erasing of a ghost-box, is difficult to handle in those terms and this rule is therefore not considered in what we are doing. Strictly speaking, we shall get a proof of standardization without this rule, but there is so little to add to take care of this rule The fact that we are forced to make ‘bricolage’ on such details is an illustration of the difficulties that arise from the absence of a real understanding of slices.

6.4. Definition (contraction of a slice). A slice β contracts into a set β' of slices (very often consisting of one slice) as follows:

- (1) (A): Axiom-contraction, like (AC) .
- (2) ($1\&/1\oplus$): Replace a configuration as in Fig. 68(a) by the one in Fig. 68(b).
- (3) ($2\&/2\oplus$): Symmetric to (2).



(a)



(b)

Fig. 68.

(4) ($1\&/2\oplus$): If in β a configuration occurs as in Fig. 69, then β contracts to \emptyset (the void set of slices).

$$\frac{\begin{array}{c} \downarrow \\ A \\ \hline A \& B \end{array} \quad \begin{array}{c} \downarrow \\ B^\perp \\ \hline A^\perp \bullet B^\perp \end{array}}{\text{CUT}}^{1\& \quad 2\oplus}$$

Fig. 69.

(5) ($2\&/1\oplus$): Symmetric to (4).

(6) ($\otimes/_$): As ($\otimes/_$ -SC).

(7) ($1/\perp$) a configuration

$$\frac{1 \quad \perp}{\text{CUT}} \quad (\text{no premises above } 1 \text{ or } \perp)$$

in β is simply cancelled.

$$\frac{\begin{array}{c} \boxed{x} \\ \downarrow \\ ?B - !C \end{array} \quad ?C^\perp}{\text{CUT}}$$

Fig. 70.

(8) ($!/W?$) a configuration as in Fig. 70 (no premise above $?C$) is replaced by

$$\begin{array}{c} ?B. \\ \uparrow \end{array}$$

(9) ($!/D?$): If β contains the configuration as in Fig. 71(a) and X is made of slices β_i (with conclusions $?B, A$), then β contracts to the set made of the slices in Fig. 71(b).

$$\frac{\begin{array}{c} \boxed{x} \\ \downarrow \\ ?B - !C \end{array} \quad \begin{array}{c} \downarrow \\ C^\perp \\ \hline ?C^\perp \end{array}}{\text{CUT}}^{D?} \quad \frac{\begin{array}{c} \boxed{\beta_i} \\ \downarrow \\ ?B - C \end{array} \quad \begin{array}{c} \downarrow \\ C^\perp \end{array}}{\text{CUT}}$$

Fig. 71.

(10) ($!/C?$): As ($!/C?-$ SC).

(11) (\wedge/\vee): If in β occurs

$$\frac{\begin{array}{c} \downarrow \\ A[a] \\ \hline \wedge a.A \end{array} \quad \begin{array}{c} \downarrow \\ A^\perp[B] \\ \hline \vee a.A^\perp \end{array}}{\text{CUT}},$$

then one contracts it by replacing a by B everywhere, and then by replacing the pattern by the cut

$$\frac{\downarrow \quad \downarrow}{\underline{A[B] \quad A^\perp[B]} \quad \text{CUT}}$$

(12) (!): Adaptation of (!-CC) to slices, details omitted.

6.5. Remark (Church-Rosser property). Using Definition 6.4, one gets a notion of reduction for sets of slices. This concept is Church-Rosser, as one can easily check.

6.6. Remark (standardization property). If, in the definition of contraction, we restrict cases (3) and (4) to the case where β cannot be normalized further, and case (8) to the case where the elements of X cannot be normalized further, we obtain the notion of standard contraction from which we derive the notion of standard reduction. The standardization property says that if a set of slices has a standard normalization into a set of slices which cannot be normalized further, then all normalization sequences starting from it are finite. This is easy to prove from the Church-Rosser property and the fact that the erasure is well-controlled in the standard case.

6.7. Remark (standardization of proof-nets). If $\beta = / \beta'$, then $\text{sl}(\beta) = / \text{sl}(\beta')$, where $\text{sl}(\cdot)$ stands for the slicing (provided (\top -CC) has not been used). This lies in the fact that the commutation rules (&-CC), (-CC), (\perp -CC), ($W?$ -CC) do nothing on the slicings. Now, this fact can easily be used to transfer standardization to proof-nets. One must work a little more to consider (\top -CC) but there are no real problems.

V. Two years of linear logic: selection from the garbage collector

This short historical note is here to explain the successive states of linear logic and, in particular, to mention possibilities that have been discarded for reasons that may seem excessive to further researchers.

V.1. The first glimpses

Linear logic first appeared after the author had been challenged by Berry and Curien to extend the coherent semantics (at that time: qualitative semantics) to the sum of types, their claim being that it was necessary to reintroduce complications that are typical for Scott domains. The answer is reproduced in an appendix of [4], and (except for the notations which are absent), one can recognize (semantically) the decomposition of the type $A \Rightarrow B$ as $!A \multimap B$ and $A \vee B$ as $!A \oplus !B$. Incidentally, the answer was found because of the insistence on interpreting all known rules for

type sum, including so-called ‘commutative rules’; commutative rules were interpreted by remarking that eliminations were linear in their main premise and so, it was sufficient to ‘linearize’ the treatment of the sum. By the way, observe that the claim that ‘eliminations are linear’ sound deliciously obsolete now since, by the existence of the involutive *nil*, there is no real distinction between introduction and elimination!

V.2. *The quantitative attempt*

After this isolated remark, nothing happened before October 1985; the subject came back to life during a working session in Torino. At that moment, the quantitative semantics was considered by the author as more promising because more weird. The quantitative semantics was showing the decomposition in a very conspicuous way, e.g., $A \Rightarrow B = \text{Int}(A).B$, suggesting to consider separately *Int* and \cdot , i.e., ‘of course’ and ‘entails’. The decompositions in the case of qualitative domains (coherent spaces) were less obvious.

The first formalism for linear logic was therefore a functional language essentially based on \neg , $!$, \otimes , \oplus , and second-order quantification, in which it was possible to make arbitrary sums (superpositions) of terms of the same type and in particular the void sum 0 of any type. Commutation rules of the sum with formation schemes expressed the linearity of everything but $!$ -introduction and terms could be normalized as sums of primitive ones. This decomposition in sum has been given up when moving to the coherent semantics, though it was a very nice feature of the calculus. In particular, it was possible to decompose a term t as $t_0 + t_1$, with t_0 the normal part of t , t_1 the part still to be executed. Also, at that time, it was possible to write an equation of the style $!A = 1 + A.!A$ and to treat normalization as a process of Taylor expansion (this formula being reminiscent of $f(dx) = f(0) + dx.f'(0)$, something like that . . .). This approach had the clear advantage of making the execution of a program never end, except in trivial cases, and this is an aspect of parallelism that has been (perhaps temporarily) lost in the formalism we have chosen in this reference version. This approach was given up because of the lack of any *logical* justification, i.e., because it was not a proof-system. For instance, the system was not making any difference between \oplus and $\&$, or accepted void terms. The coherent semantics was then introduced more and more seriously, and the logical approach became more prominent.

V.3. *The intuitionistic attempt*

As the logical framework was clarifying itself, the formulation remained strictly intuitionistic; at that moment I made attempts (with Mascari) to work out an implementation of linear sequent calculus; although the formalism was intuitionistic, the communication between sequents stayed furiously symmetric. From this moment on (January 1986), the following things appeared:

- the well-hidden connectives *nil*, *par*, *why-not*;

- the coherent semantics (restriction to binary qualitative domains to get an involutive negation) in its definite version;
- the first attempt for proof-nets.

If there are features of the quantitative attempt that we mourn, then the giving up of the intuitionistic framework is apparently completely positive. The only inconvenience is the abandonment of the functional notation which is so easy to understand; but, '*il faut souffrir pour être belle*'.

V.4. Recent developments

One of the most irritating questions was the question of *totality*: in two words, not every element of a coherent space is *robust*; e.g., most of the time, we like to exclude the empty set from the possible semantics of our terms. For this we have to speak of *total* elements, which look like the potential proofs of their type; in [5], the situation has been clarified, but in an intuitionistic framework. The newly discovered 'classical' features of linear logic made it very complicated because one would have necessarily arrived at something like: 'given two types which are linear negations of one another, one of the types is empty (nothing total) while the other is full (everything is total)'. The solution found in April '86 was 'totality with phases' and led to the Tarskian semantics of phases of August '86. Before the phase semantics, several others were tried, without complete success, and, for instance, a functional interpretation of the classical case within the intuitionistic one (with the advantage of getting rid of problems of totality) was worked out and then dumped.

Another difficult question was the way to formulate proofs: sequent calculus was appropriate, but since there is no nice normalization in sequent calculus, we got into trouble. This is the reason why, before July '86, when the fundamental results on proof-nets were obtained, the following interesting solution was considered: compute directly with the semantics. The only thing one has to be sure of is that the objects we look for are finite enough and this offers no essential difficulty. Even now, this approach retains part of its attraction.

A rule was under discussion for a while, namely the rule of MIX, which can be formulated as

$$\frac{\vdash A \quad \vdash B}{\vdash A, B}.$$

The rule says that $C \otimes D \multimap C \wp D$, which seems natural, and can be derived from $\perp \multimap \perp \otimes \perp$. Its adjunction would not change linear logic too much, however, there was a negative feeling about the rule, essentially the idea that $\vdash A, B$ means that *A* and *B* do communicate. The situation was clarified after the phase semantics appeared: semantically speaking, MIX is the requirement that \perp is closed under product. There is no clear reason to require this (without requiring $1 \in \perp$, which would be a bit too much) and MIX is now no longer under serious discussion. One of the arguments for MIX is that, without it, the type of communication considered

in proof-nets is very totalitarian: everything communicates with everything, while MIX could accept more liberal solutions, typically two non-interconnected proof-nets etc. However, what we said is true as long as \perp is not used in proof-nets since it is only for formal reasons that \perp can communicate upwards. In particular, two separate proof-nets β and β' can be put together by using \perp , the only price to pay being that another conclusion ($\perp \otimes \perp$) has been added.

But what has been under discussion most and has still not been clarified is the exact formalism for ! (and ?). The question is of great interest because behind it, the implementation of beta-conversion lies; we devote the rest of this note to give a panorama of the main lines that have been considered.

V.5. The exponentials

Already in the well-known case of lambda-calculus, there are two traditions:

- the tradition of identifying the variables, which comes from beta-conversion, when we substitute u for all occurrences of x ;
- the tradition coming from the implementation, which tries to repress or control substitution.

The first tradition rests on safe logical grounds, whereas the second one is a kind of *bricolage* with hazardous justifications.

In the first tradition, a term $t[x, y]$ will by identification yield the term $t[x, x]$ which would have also resulted from the consideration of $t[y, x]$. This is why the modelling of beta-conversion uses spaces of finite coherent sets (which can be seen in the definition of $!X$): we use a space of repetitions, but we make identifications between (x, y) and (y, x) . After many hesitations, we have eventually chosen this definition, but it is far from being perfect:

(1) Nowhere has it been said that we should model the implicative types by plain graphs (which is done with this solution). On the other hand, modelling more subtle distinctions (for instance, distinguishing between the function coming from $t[x, y]$ and the one coming from $t[y, x]$) would give a more serious basis to the second tradition.

(2) The consideration of the space of sets has a bad consequence on the formal behaviour of our semantics: functorially speaking, we lose preservation of *kernels*, which seems to indicate that a theoretical mistake has been made.

Besides the variant retained in this reference paper, several others have been considered to remedy criticisms (1) and (2):

(i) As long as linear logic was resting on quantitative ideas, the principle (inspired on the way Krivine handled beta-conversion by restricting substitution to headvariables, and on the fact that a term is linear in its headvariable) suggested to use a linear ‘first-order development’, based on the identification between $!A$ and $1 + A. !A$. The operations of identification could be seen as formal derivation or formal primitive. The interest of this approach was to propose, at the theoretical level, to replace brutal beta-conversion by iterated linear conversions.

(ii) When the quantitative approach turned out to be insufficient, the author tried to preserve this idea by means of $!A \sim 1 \& A \otimes !A$. Unfortunately, this principle is not enough to justify contraction, even if written in a form inspired on inductive definitions (projective definitions). From this attempt a variant remained, due to Lafont, where $!A$ was defined as a projective definition, and satisfies $!A \sim 1 \& A \& (!A \otimes !A)$. This idea can be modelled in terms of coherent spaces and leads to considering certain trees instead of sets.

(iii) Another possibility seriously considered is the infinitary approach: $!A = \otimes(1 \& A)$, the tensorization being made on ω copies. Here we get quite a nice theory, but we have to take care of the heavy apparatus of infinite proof-nets. For implementation we also have to restore a finitary calculus, i.e., to do some *bricolage* again. Here the semantics involves finite sequences with holes, e.g., (x_1, x_3, x_7) .⁷

The last two variants model what we have called the second tradition; i.e., they model strategies of progressive substitutions. Also, they are free from the defect we have mentioned w.r.t. kernels. However,

- (3) the isomorphism $!(A \& B) \sim !A \otimes !B$ is no longer valid (it would be valid if we were considering multisets instead of sets);
- (4) both variants do not succeed in catching the idea of first-order development.

Acknowledgment

The author is indebted to Jean-Louis Krivine, whose simultaneous work on lambda-calculus and its execution had some crucial influence on earlier versions of linear logic. Also, the author wishes to express his indebtedness to Gianfranco Mascari who introduced him to the subject of parallelism and whose influence was important for the shift to ‘classical’ framework and parallel syntax.

References

- [1] J.-Y. Girard, Une extension de l’interprétation de Gödel à l’analyse et son application à l’élimination des coupures dans l’analyse et dans la théorie des types, in: J.E. Fenstadt, ed., *Proc. 2nd Scand. Logic Symp.* (North-Holland, Amsterdam, 1971) 63-92.
- [2] J.-Y. Girard, Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur, Thèse de Doctorat d’Etat, UER de Mathématiques, Univ. de Paris VII (1972).
- [3] J.-Y. Girard, Three-valued logic and cut-elimination: the actual meaning of Takeuti’s conjecture, *Dissertationes Math.* CXXXVI (1976).
- [4] J.-Y. Girard, Normal functors, power series and lambda-calculus, *Ann. Pure Appl. Logic*, to appear.
- [5] J.-Y. Girard, The system F of variable types, fifteen years later, *Theoret. Comput. Sci.* 45(2) (1986) 159-192.
- [6] G. Kreisel and G. Takeuti, Formally self-referential propositions for cut-free classical analysis and related systems, *Dissertationes Math.* CXVIII (1974).

⁷ What remains of this idea is the Approximation Theorem of Section 5.

Gradual Typing for Functional Languages

Jeremy G. Siek

University of Colorado

siek@cs.colorado.edu

Walid Taha

Rice University

taha@rice.edu

Abstract

Static and dynamic type systems have well-known strengths and weaknesses, and each is better suited for different programming tasks. There have been many efforts to integrate static and dynamic typing and thereby combine the benefits of both typing disciplines in the same language. The flexibility of static typing can be improved by adding a type Dynamic and a typecase form. The safety and performance of dynamic typing can be improved by adding optional type annotations or by performing type inference (as in soft typing). However, there has been little formal work on type systems that allow a programmer-controlled migration between dynamic and static typing. Thatte proposed Quasi-Static Typing, but it does not statically catch all type errors in completely annotated programs. Anderson and Drossopoulou defined a nominal type system for an object-oriented language with optional type annotations. However, developing a sound, gradual type system for functional languages with structural types is an open problem.

In this paper we present a solution based on the intuition that the structure of a type may be partially known/unknown at compile-time and the job of the type system is to catch incompatibilities between the known parts of types. We define the static and dynamic semantics of a λ -calculus with optional type annotations and we prove that its type system is sound with respect to the simply-typed λ -calculus for fully-annotated terms. We prove that this calculus is type safe and that the cost of dynamism is “pay-as-you-go”.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms Languages, Performance, Theory

Keywords static and dynamic typing, optional type annotations

1. Introduction

Static and dynamic typing have different strengths, making them better suited for different tasks. Static typing provides early error detection, more efficient program execution, and better documentation, whereas dynamic typing enables rapid development and fast adaptation to changing requirements.

The focus of this paper is languages that literally provide static and dynamic typing in the same program, with the programmer control-

ling the degree of static checking by annotating function parameters with types, or not. We use the term *gradual typing* for type systems that provide this capability. Languages that support gradual typing to a large degree include Cecil [8], Boo [10], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [26], and extensions to Java proposed by Gray et al. [17], and the Bigloo [6, 36] dialect of Scheme [24]. The purpose of this paper is to provide a type-theoretic foundation for languages such as these with gradual typing.

There are numerous other ways to combine static and dynamic typing that fall outside the scope of gradual typing. Many dynamically typed languages have optional type annotations that are used to improve run-time performance but not to increase the amount of static checking. Common LISP [23] and Dylan [12, 37] are examples of such languages. Similarly, the Soft Typing of Cartwright and Fagan [7] improves the performance of dynamically typed languages but it does not statically catch type errors. At the other end of the spectrum, statically typed languages can be made more flexible by adding a Dynamic type and typecase form, as in the work by Abadi et al. [1]. However, such languages do not allow for programming in a dynamically typed style because the programmer is required to insert coercions to and from type Dynamic.

A short example serves to demonstrate the idea of gradual typing. Figure 1 shows a call-by-value interpreter for an applied λ -calculus written in Scheme extended with gradual typing and algebraic data types. The version on the left does not have type annotations, and so the type system performs little type checking and instead many tag-tests occur at run time.

As development progresses, the programmer adds type annotations to the parameters of interp, as shown on the right side of Figure 1, and the type system provides more aid in detecting errors. We use the notation ? for the dynamic type. The type system checks that the uses of env and e are appropriate: the case analysis on e is fine and so is the application of assq to x and env. The recursive calls to interp also type check and the call to apply type checks trivially because the parameters of apply are dynamic. Note that we are still using dynamic typing for the value domain of the object language. To obtain a program with complete static checking, we would introduce a datatype for the value domain and use that as the return type of interp.

Contributions We present a formal type system that supports gradual typing for functional languages, providing the flexibility of dynamically typed languages when type annotations are omitted by the programmer and providing the benefits of static checking when function parameters are annotated. These benefits include both safety and performance: type errors are caught at compile-time and values may be stored in unboxed form. That is, for statically typed portions of the program there is no need for run-time tags and tag checking.

We introduce a calculus named λ_{\rightarrow} and define its type system (Section 2). We show that this type system, when applied to fully an-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Scheme and Functional Programming 2006 September 17, Portland, OR.
Copyright © 2006 ACM [to be supplied]... \$5.00.

```

(define interp
  (λ (env e)
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n)]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")]))))

```

```

(type expr (datatype (Var ,symbol)
                      (Int ,int)
                      (App ,expr ,expr)
                      (Lam ,symbol ,expr)
                      (Succ ,expr)))
(type envty (listof (pair symbol ?)))

(define interp
  (λ ((env : envty) (e : expr))
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n)]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e)))]))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")]))))

```

Figure 1. An example of gradual typing: an interpreter with varying amounts of type annotations.

notated terms, is equivalent to that of the simply-typed lambda calculus (Theorem 1). This property ensures that for fully-annotated programs all type errors are caught at compile-time. Our type system is the first gradual type system for structural types to have this property. To show that our approach to gradual typing is suitable for imperative languages, we extend $\lambda_{\rightarrow}^?$ with ML-style references and assignment (Section 4).

We define the run-time semantics of $\lambda_{\rightarrow}^?$ via a translation to a simply-typed calculus with explicit casts, $\lambda^{(\tau)}$, for which we define a call-by-value operational semantics (Section 5). When applied to fully-annotated terms, the translation does not insert casts (Lemma 4), so the semantics exactly matches that of the simply-typed λ -calculus. The translation preserves typing (Lemma 3) and $\lambda^{(\tau)}$ is type safe (Lemma 8), and therefore $\lambda_{\rightarrow}^?$ is type safe: if evaluation terminates, the result is either a value of the expected type or a cast error, but never a type error (Theorem 2).

On the way to proving type safety, we prove Lemma 5 (Canonical Forms), which is of particular interest because it shows that the run-time cost of dynamism in $\lambda_{\rightarrow}^?$ can “pay-as-you-go”. Run-time polymorphism is restricted to values of type $?$, so for example, a value of type `int` must actually be an integer, whereas a value of type $?$ may contain an integer or a Boolean or anything at all. Compilers for $\lambda_{\rightarrow}^?$ may use efficient, unboxed, representations for values of ground and function type, achieving the performance benefits of static typing for the parts of programs that are statically typed.

The proofs of the lemmas and theorems in this paper were written in the Isar proof language [28, 42] and verified by the Isabelle proof assistant [29]. We provide proof sketches in this paper and the full proofs are available in the companion technical report [39]. The statements of the definitions (including type systems and semantics), lemmas, propositions, and theorems in this paper were automatically generated from the Isabelle files. Free variables that appear in these statements are universally quantified.

2. Introduction to Gradual Typing

The gradually-typed λ -calculus, $\lambda_{\rightarrow}^?$, is the simply-typed λ -calculus extended with a type $?$ to represent dynamic types. We present gradual typing in the setting of the simply-typed λ -calculus to reduce unnecessary distractions. However, we intend to show how gradual

typing interacts with other common language features, and as a first step combine gradual typing with ML-style references in Section 4.

Syntax of the Gradually-Typed Lambda Calculus		$e \in \lambda_{\rightarrow}^?$
Variables	$x \in \mathbb{X}$	
Ground Types	$\gamma \in \mathbb{G}$	
Constants	$c \in \mathbb{C}$	
Types	τ	$::= \gamma \mid ? \mid \tau \rightarrow \tau$
Expressions	e	$::= c \mid x \mid \lambda x : \tau . e \mid e e$ $\lambda x . e \equiv \lambda x : ? . e$

A procedure without a parameter type annotation is syntactic sugar for a procedure with parameter type $?$.

The main idea of our approach is the notion of a type whose structure may be partially known and partially unknown. The unknown portions of a type are indicated by $?$. So, for example, the type `number` $*$ $?$ is the type of a pair whose first element is of type `number` and whose second element has an unknown type. To program in a dynamically typed style, omit type annotations on parameters; they are by default assigned the type $?$. To enlist more help from the type checker, add type annotations, possibly with $?$ occurring inside the types to retain some flexibility.

The job of the static type system is to reject programs that have inconsistencies in the known parts of types. For example, the program

```
((λ (x : number) (succ x)) #t) :: reject
```

should be rejected because the type of `#t` is not consistent with the type of the parameter `x`, that is, `boolean` is not consistent with `number`. On the other hand, the program

```
((λ (x) (succ x)) #t) :: accept
```

should be accepted by the type system because the type of `x` is considered unknown (there is no type annotation) and therefore not within the realm of static checking. Instead, the type error will be caught at run-time (as is typical of dynamically typed languages), which we describe in Section 5.

As usual things become more interesting with first class procedures. Consider the following example of mapping a procedure over a list.

```
map : (number → number) * number list → number list
(map (λ (x) (succ x)) (list 1 2 3)) ;; accept
```

The `map` procedure is expecting a first argument whose type is `number → number` but the argument $(\lambda(x)(\text{succ } x))$ has type $? \rightarrow \text{number}$. We would like the type system to accept this program, so how should we define consistency for procedure types? The intuition is that we should require the known portions of the two types to be equal and ignore the unknown parts. There is a useful analogy with the mathematics of partial functions: two partial functions are *consistent* when every elements that is in the domain of both functions is mapped to the same result. This analogy can be made formal by considering types as trees [32].



Trees can be represented as partial functions from paths to node labels, where a path is a sequence of edge labels: $[l_1, \dots, l_n]$. The above two trees are the following two partial functions f and g . We interpret unknown portions of a type simply as places where the partial function is undefined. So, for example, g is undefined for the path $[cod]$.

$$\begin{aligned} f([]) &= \rightarrow \\ f([dom]) &= \text{number} \\ f([cod]) &= \text{number} \end{aligned}$$

$$\begin{aligned} g([]) &= \rightarrow \\ g([dom]) &= \text{number} \end{aligned}$$

The partial functions f and g are consistent because they produce the same output for the inputs $[]$ and $[dom]$.

We axiomatize the consistency relation \sim on types with the following definition.

Type Consistency	$\tau \sim \tau$
(CREFL) $\tau \sim \tau$	(CFUN) $\frac{\sigma_1 \sim \tau_1 \quad \sigma_2 \sim \tau_2}{\sigma_1 \rightarrow \sigma_2 \sim \tau_1 \rightarrow \tau_2}$
(CUNR) $\tau \sim ?$	(CUNL) $? \sim \tau$

The type consistency relation is reflexive and symmetric but not transitive (just like consistency of partial functions).

Proposition 1.

- $\tau \sim \tau$
- If $\sigma \sim \tau$ then $\tau \sim \sigma$.
- $\neg (\forall \tau_1 \tau_2 \tau_3. \tau_1 \sim \tau_2 \wedge \tau_2 \sim \tau_3 \longrightarrow \tau_1 \sim \tau_3)$

Our gradual type system is shown in Figure 2. The environment Γ is a function from variables to optional types ($\lfloor \tau \rfloor$ or \perp). The type system is parameterized on a signature Δ that assigns types to constants. The rules for variables, constants, and functions are standard. The first rule for function application (GAPP1) handles the case when the function type is unknown. The argument may have any type and the resulting type of the application is unknown. The second rule for function application (GAPP2) handles when

Figure 2. A Gradual Type System

	$\boxed{\Gamma \vdash_G e : \tau}$
(GVAR)	$\frac{}{\Gamma x = \lfloor \tau \rfloor} \Gamma \vdash_G x : \tau$
(GCONST)	$\frac{\Delta c = \tau}{\Gamma \vdash_G c : \tau}$
(GLAM)	$\frac{\Gamma(x \mapsto \sigma) \vdash_G e : \tau}{\Gamma \vdash_G \lambda x:\sigma. e : \sigma \rightarrow \tau}$
(GAPP1)	$\frac{\Gamma \vdash_G e_1 : ? \quad \Gamma \vdash_G e_2 : \tau_2}{\Gamma \vdash_G e_1 e_2 : ?}$
(GAPP2)	$\frac{\Gamma \vdash_G e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash_G e_2 : \tau_2 \quad \tau_2 \sim \tau}{\Gamma \vdash_G e_1 e_2 : \tau'}$

the function type is known and allows an argument whose type is consistent with the function's parameter type.

Relation to the untyped λ -calculus We would like our gradual type system to accept all terms of the untyped λ -calculus (all unannotated terms), but it is not possible to simultaneously achieve this and provide type safety for fully-annotated terms. For example, suppose there is a constant `succ` with type `number → number`. The term $(\text{succ } "hi")$ has no type annotations but it is also fully annotated because there are no function parameters to annotate. The type system must either accept or reject this program. We choose to reject. Of course, if `succ` were given the type $? \rightarrow ?$ then $(\text{succ } "hi")$ would be accepted. In any event, our gradual type system provides the same expressiveness as the untyped λ -calculus. The following translation converts any λ -term into an observationally equivalent well-typed term of λ_\rightarrow .

$$\begin{aligned} \llbracket c \rrbracket &= c \\ \llbracket x \rrbracket &= x \\ \llbracket \lambda x. e \rrbracket &= \lambda x. \llbracket e \rrbracket \\ \llbracket e_1 e_2 \rrbracket &= ((\lambda x. x) \llbracket e_1 \rrbracket) \llbracket e_2 \rrbracket \end{aligned}$$

Relation to the simply-typed λ -calculus Let λ_\rightarrow denote the terms of the simply-typed λ -calculus and let $\Gamma \vdash_\rightarrow e : \tau$ stand for the standard typing judgment of the simply-typed λ -calculus. For terms in λ_\rightarrow our gradual type system is equivalent to simple typing.

Theorem 1 (Equivalence to simple typing for fully-annotated terms). If $e \in \lambda_\rightarrow$ then $\emptyset \vdash_G e : \tau = \emptyset \vdash_\rightarrow e : \tau$.

Proof Sketch. The rules for our gradual type system are the same as for the STLC if one removes the rules that mention $?$. The type compatibility relation collapses to type equality once all rules involving $?$ are removed. \square

A direct consequence of this equivalence is that our gradual type system catches the same static errors as the type system for λ_\rightarrow .

Corollary 1 (Full static error detection for fully-annotated terms). If $e \in \lambda_\rightarrow$ and $\not\models \tau. \emptyset \vdash_\rightarrow e : \tau$ then $\not\models \tau'. \emptyset \vdash_G e : \tau'$. (This is just the contrapositive of soundness.)

Before describing the run-time semantics of $\lambda_{\rightarrow}^?$ we compare our type system for $\lambda_{\rightarrow}^?$ with an alternative design based on subtyping.

3. Comparison with Quasi-Static Typing

Our first attempt to define a gradual type system was based on Thatte's quasi-static types [40]. Thatte uses a standard subtyping relation $<:$ with a top type Ω to represent the dynamic type. As before, the meta-variable γ ranges over ground types such as **number** and **boolean**.

Subtyping rules.

$$\frac{}{\gamma <: \gamma} \quad \frac{}{\tau <: \Omega} \quad \frac{\sigma_1 <: \tau_1 \quad \tau_2 <: \sigma_2}{\tau_1 \rightarrow \tau_2 <: \sigma_1 \rightarrow \sigma_2} \quad \frac{}{\tau <: \tau'}$$

The quasi-static type system includes the usual subsumption rule.

$$\text{QSUB } \frac{\Gamma \vdash e : \tau \quad \tau <: \sigma}{\Gamma \vdash e : \sigma}$$

Subsumption allows programs such as the following to type check by allowing implicit up-casts. The value `#t` of type **boolean** is up-cast to Ω , the type of the parameter x .

$((\lambda (x) ...) \#t) ;;$ ok, **boolean** $<: \Omega$

However, the subsumption rule will not allow the following program to type check. The addition operator expects type **number** but gets an argument of type Ω .

$(\lambda (x) (\text{succ } x))$

Thatte's solution for this is to also allow an implicit down-cast in the (QAPP2) rule for function application.

$$(\text{QAPP2}) \frac{\Gamma \vdash e_1 : \sigma \rightarrow \sigma' \quad \Gamma \vdash e_2 : \tau \quad \sigma <: \tau}{\Gamma \vdash (e_1 e_2) : \sigma'}$$

Unfortunately, the subsumption rule combined with (QAPP2) allows too many programs to type check for our taste. For example, we can build a typing derivation for the following program, even though it was rejected by our gradual type system.

$((\lambda (x : \text{number}) (\text{succ } x)) \#t)$

The subsumption rule allows `#t` to be implicitly cast to Ω and then the above rule for application implicitly casts Ω down to **number**.

To catch errors such as these, Thatte added a second phase to the type system called plausibility checking. This phase rewrites the program by collapsing sequences of up-casts and down-casts and signals an error if it encounters a pair of casts that together amount to a “stupid cast”[22], that is, casts that always fail because the target is incompatible with the subject.

Figure 3 shows Thatte's Quasi-Static type system. The judgment $\Gamma \vdash e \Rightarrow e' : \tau$ inserts up-casts and down-casts and the judgment $e \rightsquigarrow e'$ collapses sequences of casts and performs plausibility checking. The type system is parameterized on the function Δ mapping constants to types. The environment Γ is a function from variables to optional types ($[\tau]$ or \perp).

Subsumption rules are slippery, and even with the plausibility checks the type system fails to catch many errors. For example, there is still a derivation for the program

$((\lambda (x : \text{number}) (\text{succ } x)) \#t)$

The reason is that both the operator and operand may be implicitly up-cast to Ω . The rule (QAPP1) then down-casts the operator to $\Omega \rightarrow \Omega$. Plausibility checking succeeds because there is a

Figure 3. Thatte's Quasi-Static Typing.

		$\Gamma \vdash e \Rightarrow e' : \tau$
(QVAR)	$\frac{}{\Gamma x = [\tau]}$	$\Gamma \vdash x \Rightarrow x : \tau$
(QCONST)	$\frac{\Delta c = \tau}{\Gamma \vdash c \Rightarrow c : \tau}$	
(QLAM)	$\frac{\Gamma, x : \tau \vdash e \Rightarrow e' : \sigma}{\Gamma \vdash (\lambda x : \tau. e) \Rightarrow (\lambda x : \tau. e') : \tau \rightarrow \sigma}$	
(QSUB)	$\frac{\Gamma \vdash e \Rightarrow e' : \tau \quad \tau <: \sigma}{\Gamma \vdash e \Rightarrow e' \uparrow_{\tau}^{\sigma} : \sigma}$	
(QAPP1)	$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \Omega \quad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau}{\Gamma \vdash (e_1 e_2) \Rightarrow ((e'_1 \downarrow_{\tau}^{\Omega}) e'_2) : \Omega}$	$\Gamma \vdash e \rightsquigarrow e'$
(QAPP2)	$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \sigma \rightarrow \sigma' \quad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau \quad \sigma <: \tau}{\Gamma \vdash (e_1 e_2) \Rightarrow (e'_1 (e'_2 \downarrow_{\sigma}^{\tau})) : \sigma'}$	
		$e \downarrow_{\tau}^{\tau} \rightsquigarrow e \quad e \uparrow_{\tau}^{\tau} \rightsquigarrow e$
		$e \downarrow_{\sigma}^{\tau} \downarrow_{\mu}^{\sigma} \rightsquigarrow e \downarrow_{\mu}^{\tau} \quad e \uparrow_{\mu}^{\sigma} \uparrow_{\sigma}^{\tau} \rightsquigarrow e \uparrow_{\mu}^{\tau}$
		$\frac{\mu = \tau \sqcap \nu}{e \downarrow_{\tau}^{\sigma} \downarrow_{\nu}^{\sigma} \rightsquigarrow e \downarrow_{\mu}^{\tau} \uparrow_{\mu}^{\nu}} \quad \frac{\exists \mu. \mu = \tau \sqcap \nu}{e \uparrow_{\tau}^{\sigma} \downarrow_{\nu}^{\sigma} \rightsquigarrow \text{wrong}}$

greatest lower bound of **number** \rightarrow **number** and $\Omega \rightarrow \Omega$, which is $\Omega \rightarrow$ **number**. So the quasi-static system fails to statically catch the type error.

As noted by Oliart [30], Thatte's quasi-static type system does not correspond to his type checking *algorithm* (Theorem 7 of [40] is incorrect). Thatte's type checking algorithm does not suffer from the above problems because the algorithm does not use the subsumption rule and instead performs all casting at the application rule, disallowing up-casts to Ω followed by arbitrary down-casts. Oliart defined a simple syntax-directed type system that is equivalent to Thatte's algorithm, but did not state or prove any of its properties. We initially set out to prove type safety for Oliart's subtype-based type system, but then realized that the consistency relation provides a much simpler characterization of when implicit casts should be allowed.

At first glance it may seem odd to use a symmetric relation such as consistency instead of an anti-symmetric relation such as subtyping. There is an anti-symmetric relation that is closely related to consistency, the usual partial ordering relation for partial functions: $f \sqsubseteq g$ if the graph of f is a subset of the graph of g . (Note that the direction is flipped from that of the subtyping relation $<:$, where greater means less information.) A cast from τ to σ , where $\sigma \sqsubseteq \tau$, always succeeds at run-time as we are just hiding type information by replacing parts of a type with $?$. On the other hand, a cast from σ to τ may fail because the run-time type of the value may not be consistent with τ . The main difference between \sqsubseteq and $<:$ is that \sqsubseteq is covariant for the domain of a procedure type, whereas $<:$ is contra-variant for the domain of a procedure type.

Figure 4. Type Rules for References

	$\boxed{\Gamma \vdash_G e : \tau}$
(GREF)	$\frac{\Gamma \vdash_G e : \tau}{\Gamma \vdash_G \text{ref } e : \text{ref } \tau}$
(GDEREFL)	$\frac{\Gamma \vdash_G e : ?}{\Gamma \vdash_G !e : ?}$
(GDEREF2)	$\frac{\Gamma \vdash_G e : \text{ref } \tau}{\Gamma \vdash_G !e : \tau}$
(GASSIGN1)	$\frac{\Gamma \vdash_G e_1 : ? \quad \Gamma \vdash_G e_2 : \tau}{\Gamma \vdash_G e_1 \leftarrow e_2 : \text{ref } \tau}$
(GASSIGN2)	$\frac{\Gamma \vdash_G e_1 : \text{ref } \tau \quad \Gamma \vdash_G e_2 : \sigma \quad \sigma \sim \tau}{\Gamma \vdash_G e_1 \leftarrow e_2 : \text{ref } \tau}$

4. Gradual Typing and References

It is often challenging to integrate type system extensions with imperative features such as references with assignment. In this section we extend the calculus to include ML-style references. The following grammar shows the additions to the syntax.

Adding references to $\lambda^?$

$$\begin{array}{ll} \text{Types} & \tau ::= \dots \mid \text{ref } \tau \\ \text{Expressions} & e ::= \dots \mid \text{ref } e \mid !e \mid e \leftarrow e \end{array}$$

The form $\text{ref } e$ creates a reference cell and initializes it with the value that results from evaluating expression e . The dereference form $!e$ evaluates e to the address of a location in memory (hopefully) and returns the value stored there. The assignment form $e \leftarrow e$ stores the value from the right-hand side expression in the location given by the left-hand side expression.

Figure 4 shows the gradual typing rules for these three new constructs. In the (GASSIGN2) we allow the type of the right-hand side to differ from the type in the left-hand's reference, but require the types to be compatible. This is similar to the (GAPP2) rule for function application.

We do not change the definition of the consistency relation, which means that references types are invariant with respect to consistency. The reflexive axiom $\tau \sim \tau$ implies that $\text{ref } \tau \sim \text{ref } \tau$. The situation is analogous to that of the combination of references with subtyping [32]: allowing variance under reference types compromises type safety. The following program demonstrates how a covariant rule for reference types would allow type errors to go unnoticed by the type system.

```
let r1 = ref ( $\lambda$  y. y) in
let r2 : ref ? = r1 in
  r2  $\leftarrow$  1;
  !r1 2
```

The reference $r1$ is initialized with a function, and then $r2$ is aliased to $r1$, using the covariance to allow the change in type to $\text{ref } ?$. We can then write an integer into the cell pointed to by $r2$ (and by $r1$). The subsequent attempt to apply the contents of $r1$ as if it were a function fails at runtime.

5. Run-time semantics

We define the semantics for $\lambda^?$ in two steps. We first define a cast insertion translation from $\lambda^?$ to an intermediate language with explicit casts which we call $\lambda^{\langle \tau \rangle}$. We then define a call-by-value operational semantics for $\lambda^{\langle \tau \rangle}$. The explicit casts have the syntactic form $\langle \tau \rangle e$ where τ is the target type. When e evaluates to v , the cast will check that the type of v is consistent with τ and then produce a value based on v that has the type τ . If the type of v is inconsistent with τ , the cast produces a *CastError*. The intuition behind this kind of cast is that it reinterprets a value to have a different type either by adding or removing type information.

The syntax of $\lambda^{\langle \tau \rangle}$ extends that of $\lambda^?$ by adding a cast expression.

Syntax of the intermediate language.

$$e \in \lambda^{\langle \tau \rangle}$$

$$\text{Expressions } e ::= \dots \mid \langle \tau \rangle e$$

5.1 Translation to $\lambda^{\langle \tau \rangle}$.

The cast insertion judgment, defined in Figure 5, has the form $\Gamma \vdash e \Rightarrow e' : \tau$ and mimics the structure of our gradual typing judgment of Figure 2. It is trivial to show that these two judgments accept the same set of terms. We presented the gradual typing judgment separately to provide an uncluttered *specification* of well-typed terms. In Figure 5, the rules for variables, constants, and functions are straightforward. The first rule for application (CAPP1) handles the case when the function has type $?$ and inserts a cast to $\tau_2 \rightarrow ?$ where τ_2 is the argument's type. The second rule for application (CAPP2) handles the case when the function's type is known and the argument type differs from the parameter type, but is consistent. In this case the argument is cast to the parameter type τ . We could have instead cast the function; the choice was arbitrary. The third rule for application (CAPP3) handles the case when the function type is known and the argument's type is identical to the parameter type. No casts are needed in this case. The rules for reference assignment are similar to the rules for application. However, for CASSIGN2 the choice to cast the argument and not the reference is because we need references to be invariant to preserve type soundness.

Next we define a type system for the intermediate language $\lambda^{\langle \tau \rangle}$. The typing judgment has the form $\Gamma | \Sigma \vdash e : \tau$. The Σ is a store typing: it assigns types to memory locations. The type system, defined in Figure 6, extends the STLC with a rule for explicit casts. The rule (TCAST) requires the expression e to have a type consistent with the target type τ .

The inversion lemmas for $\lambda^{\langle \tau \rangle}$ are straightforward.

Lemma 1 (Inversion on typing rules.).

1. If $\Gamma | \Sigma \vdash x : \tau$ then $\Gamma x = [\tau]$.
2. If $\Gamma | \Sigma \vdash c : \tau$ then $\Delta c = \tau$.
3. If $\Gamma | \Sigma \vdash \lambda x : \sigma. e : \tau$ then $\exists \tau'. \tau = \sigma \rightarrow \tau'$.
4. If $\Gamma | \Sigma \vdash e_1 e_2 : \tau'$ then $\exists \tau. \Gamma | \Sigma \vdash e_1 : \tau \rightarrow \tau' \wedge \Gamma | \Sigma \vdash e_2 : \tau$.
5. If $\Gamma | \Sigma \vdash \langle \sigma \rangle e : \tau$ then $\exists \tau'. \Gamma | \Sigma \vdash e : \tau' \wedge \sigma = \tau \wedge \tau' \sim \sigma$.
6. If $\Gamma | \Sigma \vdash \text{ref } e : \text{ref } \tau$ then $\Gamma | \Sigma \vdash e : \tau$.
7. If $\Gamma | \Sigma \vdash !e : \tau$ then $\Gamma | \Sigma \vdash e : \text{ref } \tau$.
8. If $\Gamma | \Sigma \vdash e_1 \leftarrow e_2 : \text{ref } \tau$ then $\Gamma | \Sigma \vdash e_1 : \text{ref } \tau \wedge \Gamma | \Sigma \vdash e_2 : \tau$.

Figure 5. Cast Insertion

	$\boxed{\Gamma \vdash e \Rightarrow e' : \tau}$
(CVAR)	$\frac{\Gamma x = [\tau]}{\Gamma \vdash x \Rightarrow x : \tau}$
(CCONST)	$\frac{\Delta c = \tau}{\Gamma \vdash c \Rightarrow c : \tau}$
(CLAM)	$\frac{\Gamma(x \mapsto \sigma) \vdash e \Rightarrow e' : \tau}{\Gamma \vdash \lambda x:\sigma. e \Rightarrow \lambda x:\sigma. e' : \sigma \rightarrow \tau}$
(CAPP1)	$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : ? \quad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2}{\Gamma \vdash e_1 e_2 \Rightarrow (\langle \tau_2 \rightarrow ? \rangle e'_1) e'_2 : ?}$
(CAPP2)	$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2 \quad \tau_2 \neq \tau \quad \tau_2 \sim \tau}{\Gamma \vdash e_1 e_2 \Rightarrow e'_1 (\langle \tau \rangle e'_2) : \tau'}$
(CAPP3)	$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau}{\Gamma \vdash e_1 e_2 \Rightarrow e'_1 e'_2 : \tau'}$
(CREF)	$\frac{\Gamma \vdash e \Rightarrow e' : \tau}{\Gamma \vdash \text{ref } e \Rightarrow \text{ref } e' : \text{ref } \tau}$
(CDEREF1)	$\frac{\Gamma \vdash e \Rightarrow e' : ?}{\Gamma \vdash !e \Rightarrow !(\langle \text{ref} ? \rangle e') : ?}$
(CDEREF2)	$\frac{\Gamma \vdash e \Rightarrow e' : \text{ref } \tau}{\Gamma \vdash !e \Rightarrow e' : \tau}$
(CASSIGN1)	$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : ? \quad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow (\langle \text{ref } \tau_2 \rangle e'_1) \leftarrow e'_2 : \text{ref } \tau_2}$
(CASSIGN2)	$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \text{ref } \tau \quad \Gamma \vdash e_2 \Rightarrow e'_2 : \sigma \quad \sigma \neq \tau \quad \sigma \sim \tau}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow e'_1 \leftarrow (\langle \tau \rangle e'_2) : \text{ref } \tau}$
(CASSIGN3)	$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \text{ref } \tau \quad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow e'_1 \leftarrow e'_2 : \text{ref } \tau}$

Proof Sketch. They are proved by case analysis on the type rules. \square

The type system for $\lambda^{\langle \tau \rangle}$ is deterministic: it assigns a unique type to an expression given a fixed environment.

Lemma 2 (Unique typing). If $\Gamma \mid \Sigma \vdash e : \tau$ and $\Gamma \mid \Sigma \vdash e : \tau'$ then $\tau = \tau'$.

Proof Sketch. The proof is by induction on the typing derivation and uses the inversion lemmas. \square

The cast insertion translation, if successful, produces well-typed terms of $\lambda^{\langle \tau \rangle}$.

Lemma 3. If $\Gamma \vdash e \Rightarrow e' : \tau$ then $\Gamma \mid \emptyset \vdash e' : \tau$.

Figure 6. Type system for the intermediate language $\lambda^{\langle \tau \rangle}$

	$\boxed{\Gamma \mid \Sigma \vdash e : \tau}$
(TVar)	$\frac{\Gamma x = [\tau]}{\Gamma \mid \Sigma \vdash x : \tau}$
(TConst)	$\frac{\Delta c = \tau}{\Gamma \mid \Sigma \vdash c : \tau}$
(TLam)	$\frac{\Gamma(x \mapsto \sigma) \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \lambda x:\sigma. e : \sigma \rightarrow \tau}$
(TApp)	$\frac{\Gamma \mid \Sigma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash e_1 e_2 : \tau'}$
(TCast)	$\frac{\Gamma \mid \Sigma \vdash e : \sigma \quad \sigma \sim \tau}{\Gamma \mid \Sigma \vdash \langle \tau \rangle e : \tau}$
(TRef)	$\frac{\Gamma \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \text{ref } e : \text{ref } \tau}$
(TDeref)	$\frac{\Gamma \mid \Sigma \vdash e : \text{ref } \tau}{\Gamma \mid \Sigma \vdash \text{!}e : \tau}$
(TAssign)	$\frac{\Gamma \mid \Sigma \vdash e_1 : \text{ref } \tau \quad \Gamma \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash e_1 \leftarrow e_2 : \text{ref } \tau}$
(TLoc)	$\frac{\Sigma l = [\tau]}{\Gamma \mid \Sigma \vdash l : \text{ref } \tau}$

Proof Sketch. The proof is by induction on the cast insertion derivation. \square

When applied to terms of λ_{\rightarrow} , the translation is the identity function, i.e., no casts are inserted.¹

Lemma 4. If $\emptyset \vdash e \Rightarrow e' : \tau$ and $e \in \lambda_{\rightarrow}$ then $e = e'$.

Proof Sketch. The proof is by induction on the cast insertion derivation. \square

When applied to terms of the untyped λ -calculus, the translation inserts just those casts necessary to prevent type errors from occurring at run-time, such as applying a non-function.

5.2 Run-time semantics of $\lambda^{\langle \tau \rangle}$.

The following grammar describes the results of evaluation: the result is either a value or an error, where values are either a simple value (variables, constants, functions, and locations) or a simple value enclosed in a single cast, which serves as a syntactical representation of boxed values.

¹This lemma is for closed terms (this missing Γ means an empty environment). A similar lemma is true of open terms, but we do not need the lemma for open terms and the statement is more complicated because there are conditions on the environment.

Values, Errors, and Results

Locations	$l \in \mathbb{L}$
Simple Values	$s \in \mathbb{S} ::= x \mid c \mid \lambda x : \tau.e \mid l$
Values	$v \in \mathbb{V} ::= s \mid \langle ? \rangle s$
Errors	$\varepsilon ::= \text{CastError} \mid \text{TypeError} \mid \text{KillError}$
Results	$r ::= v \mid \varepsilon$

It is useful to distinguish two different kinds of run-time type errors. In weakly typed languages, type errors result in undefined behavior, such as causing a segmentation fault or allowing a hacker to create a buffer overflow. We model this kind of type error with *TypeError*. In strongly-typed dynamic languages, there may still be type errors, but they are caught by the run-time system and do not cause undefined behavior. They typically cause the program to terminate or else raise an exception. We model this kind of type error with *CastError*. The *KillError* is a technicality pertaining to the type safety proof that allows us to prove a form of “progress” in the setting of a big-step semantics.

We define simple function values (*SimpleFunVal*) to contain lambda abstractions and functional constants (such as *succ*), and function values (*FunVal*) include simple function values and simple function values cast to $\langle ? \rangle$.

As mentioned in Section 1, the Canonical Forms Lemma is of particular interest due to its implications for performance. When an expression has either ground or function type (not $\langle ? \rangle$) the kind of resulting value is fixed, and a compiler may use an efficient unboxed representation. For example, if an expression has type *int*, then it will evaluate to a value of type *int* (by the forthcoming Soundness Lemma 8) and then the Canonical Forms Lemma tells us that the value must be an integer.

Lemma 5 (Canonical Forms).

- If $\emptyset \mid \Sigma \vdash v : \text{int}$ and $v \in \mathbb{V}$ then $\exists n. v = n$.
- If $\emptyset \mid \Sigma \vdash v : \text{bool}$ and $v \in \mathbb{V}$ then $\exists b. v = b$.
- If $\emptyset \mid \Sigma \vdash v : \langle ? \rangle$ and $v \in \mathbb{V}$ then $\exists v'. v = \langle ? \rangle v' \wedge v' \in \mathbb{S}$.
- If $\emptyset \mid \Sigma \vdash v : \tau \rightarrow \tau'$ and $v \in \mathbb{V}$ then $v \in \text{SimpleFunVal}$.
- If $\emptyset \mid \Sigma \vdash v : \text{ref } \tau$ and $v \in \mathbb{V}$ then $\exists l. v = l \wedge \Sigma l = \lfloor \tau \rfloor$.

Proof Sketch. They are proved using the inversion lemmas and case analysis on values. \square

We define the run-time semantics for $\lambda \downarrow$ in big-step style with substitution and not environments. Substitution, written $[x := e]e$, is formalized in the style of Curry [3], where bound variables are α -renamed during substitution to avoid the capture of free variables.

The evaluation judgment has the form $e \hookrightarrow_n r$, where e evaluates to the result r with a derivation depth of n . The derivation depth is used to force termination so that derivations can be constructed for otherwise non-terminating programs [11]. The n -depth evaluation allows Lemma 8 (Soundness) to distinguish between terminating and non-terminating programs. We will say more about this when we get to Lemma 8.

The evaluation rules, shown in Figures 7 and 8, are the standard call-by-value rules for the λ -calculus [33] with additional rules for casts and a special termination rule. We parameterize the semantics over the function δ which defines the behavior of functional constants and is used in rule (EDELTA). The helper function *unbox* removes an enclosing cast from a value, if there is one.

$$\begin{array}{ll} \text{unbox } s &= s \\ \text{unbox } (\langle \tau \rangle s) &= s \end{array}$$

The evaluation rules treat the cast expression like a boxed, or tagged, value. It is straightforward to define a lower-level semantics that explicitly tags every value with its type (the full type, not just the top level constructor) and then uses these type representations instead of the typing judgment $\emptyset \mid \emptyset \vdash \text{unbox } v : \tau$, as in the rule (ECSTG).

There is a separate cast rule for each kind of target type. The rule (ECSTG) handles the case of casting to a ground type. The cast is removed provided the run-time type exactly matches the target type. The rule (ECSTF) handles the case of casting to a function type. If the run-time type is consistent with the target type, the cast is removed and the inner value is wrapped inside a new function that inserts casts to produce a well-typed value of the appropriate type. This rule is inspired by the work on semantic casts [13, 14, 15], though the rule may look slightly different because the casts used in this paper are annotated with the target type only and not also with the source type. The rule (ECSTR) handles the case of casting to a reference type. The run-time type must exactly match the target type. The rule (ECSTU) handles the case of casting to $\langle ? \rangle$ and ensures that nested casts are collapsed to a single cast. The rule (ECSTE) handles the case when the run-time type is not consistent with the target type and produces a *CastError*. Because the target types of casts are static, the cast form could be replaced by a cast for each type, acting as injection to $\langle ? \rangle$ and projection to ground and function types. However, this would complicate the rules, especially the rule for casting to a function type.

The rule (EKILL) terminates evaluation when the derivation depth counter reaches zero.

5.3 Examples

Consider once again the following program and assume the *succ* constant has the type **number** \rightarrow **number**.

$$((\lambda (x) (\text{succ } x)) \#t)$$

The cast insertion judgement transforms this term into the following term.

$$((\lambda (x : \langle ? \rangle) (\text{succ } \langle \text{number} \rangle x)) \langle ? \rangle \#t)$$

Evaluation then proceeds, applying the function to its argument, substituting $\langle ? \rangle \#t$ for x . \square

$$(\text{succ } \langle \text{number} \rangle \langle ? \rangle \#t)$$

The type of $\#t$ is **boolean**, which is not consistent with **number**, so the rule (ECSTE) applies and the result is a cast error.

$$\text{CastError}$$

Next, we look at an example that uses first-class functions.

$$((\lambda (f : ? \rightarrow \text{number}) (f 1)) \\ (\lambda (x : \text{number}) (\text{succ } x)))$$

Cast insertion results in the following program.

$$((\lambda (f : ? \rightarrow \text{number}) (f \langle ? \rangle 1)) \\ (\langle ? \rangle \#t (\lambda (x : \text{number}) (\text{succ } x))))$$

We apply the cast to the function, creating a wrapper function.

$$((\lambda (f : ? \rightarrow \text{number}) (f \langle ? \rangle 1)) \\ (\lambda (z : ?) \langle \text{number} \rangle ((\lambda (x : \text{number}) (\text{succ } x)) \langle \text{number} \rangle z)))$$

Function application results in the following

Figure 7. Evaluation

	$e \mu \hookrightarrow_n r \mu$
Casting	
(ECSTG)	$\frac{e \mu \hookrightarrow_n v \mu' \quad \emptyset \Sigma \vdash \text{unbox } v : \gamma}{\langle \gamma \rangle e \mu \hookrightarrow_{n+1} \text{unbox } v \mu'}$
(ECSTF)	$\frac{e \mu \hookrightarrow_n v \mu' \quad \emptyset \Sigma \vdash \text{unbox } v : \tau \rightarrow \tau' \quad \tau \rightarrow \tau' \sim \sigma \rightarrow \sigma' \quad z = \max v + 1}{\langle \sigma \rangle e \mu \hookrightarrow_{n+1} \lambda z : \sigma. (\langle \sigma' \rangle (\text{unbox } v (\langle \tau \rangle z))) \mu'}$
(ECSTR)	$\frac{e \mu \hookrightarrow_n v \mu' \quad \emptyset \Sigma \vdash \text{unbox } v : \text{ref } \tau \quad \langle \text{ref } \tau \rangle e \mu \hookrightarrow_{n+1} \text{unbox } v \mu'}{\langle \text{ref } \tau \rangle e \mu \hookrightarrow_{n+1} \text{unbox } v \mu'}$
(ECSTU)	$\frac{e \mu \hookrightarrow_n v \mu'}{\langle ? \rangle e \mu \hookrightarrow_{n+1} \langle ? \rangle \text{unbox } v \mu'}$
Functions and constants	
(ELAM)	$\frac{0 < n}{\lambda x : \tau. e \mu \hookrightarrow_n \lambda x : \tau. e \mu}$
(EAPP)	$\frac{e_1 \mu_1 \hookrightarrow_n \lambda x : \tau. e_3 \mu_2 \quad e_2 \mu_2 \hookrightarrow_n v_2 \mu_3 \quad [x := v_2] e_3 \mu_3 \hookrightarrow_n v_3 \mu_4}{e_1 e_2 \mu_1 \hookrightarrow_{n+1} v_3 \mu_4}$
(ECONST)	$\frac{0 < n}{c \mu \hookrightarrow_n c \mu}$
(EDELT)	$\frac{e_1 \mu_1 \hookrightarrow_n c_1 \mu_2 \quad e_2 \mu_2 \hookrightarrow_n c_2 \mu_3}{e_1 e_2 \mu_1 \hookrightarrow_{n+1} \delta c_1 c_2 \mu_3}$
References	
(EREF)	$\frac{e \mu \hookrightarrow_n v \mu' \quad l \notin \text{dom } \mu'}{\text{ref } e \mu \hookrightarrow_{n+1} l \mu' (l \mapsto v)}$
(EDEREFT)	$\frac{e \mu \hookrightarrow_n l \mu' \quad \mu' l = [v]}{!e \mu \hookrightarrow_{n+1} v \mu'}$
(EASSIGN)	$\frac{e_1 \mu_1 \hookrightarrow_n l \mu_2 \quad e_2 \mu_2 \hookrightarrow_n v \mu_3}{e_1 \leftarrow e_2 \mu_1 \hookrightarrow_{n+1} l \mu_3 (l \mapsto v)}$
(ELOC)	$\frac{0 < n}{l \mu \hookrightarrow_n l \mu}$

Figure 8. Evaluation (Errors)

(ECSTE)	$\frac{\emptyset \Sigma \vdash \text{unbox } v : \sigma \quad (\sigma, \tau) \notin \text{op} \sim}{\langle \tau \rangle e \mu \hookrightarrow_{n+1} \text{CastError} \mu'}$
(EKILL)	$e \mu \hookrightarrow_0 \text{KillError} \mu$
(EVART)	$\frac{0 < n}{x \mu \hookrightarrow_n \text{TypeError} \mu}$
(EAPPT)	$\frac{e_1 \mu \hookrightarrow_n v_1 \mu' \quad v_1 \notin \text{FunVal}}{e_1 e_2 \mu \hookrightarrow_{n+1} \text{TypeError} \mu'}$
(ECSTP)	$\frac{e \mu \hookrightarrow_n \varepsilon \mu'}{\langle \tau \rangle e \mu \hookrightarrow_{n+1} \varepsilon \mu'}$
(EAPP1)	$\frac{e_1 \mu \hookrightarrow_n \varepsilon \mu'}{e_1 e_2 \mu \hookrightarrow_{n+1} \varepsilon \mu'}$
(EAPP2)	$\frac{e_1 \mu_1 \hookrightarrow_n v_1 \mu_2 \quad v_1 \in \text{FunVal}}{e_1 e_2 \mu_1 \hookrightarrow_{n+1} \varepsilon \mu_3}$
(EAPP3)	$\frac{e_1 \mu_1 \hookrightarrow_n \lambda x : \tau. e_3 \mu_2 \quad e_2 \mu_2 \hookrightarrow_n v_2 \mu_3 \quad [x := v_2] e_3 \mu_3 \hookrightarrow_n \varepsilon \mu_4}{e_1 e_2 \mu_1 \hookrightarrow_{n+1} \varepsilon \mu_4}$
(EREF)	$\frac{e \mu \hookrightarrow_n \varepsilon \mu'}{\text{ref } e \mu \hookrightarrow_{n+1} \varepsilon \mu'}$
(EDEREFP)	$\frac{e \mu \hookrightarrow_n \varepsilon \mu'}{!e \mu \hookrightarrow_{n+1} \varepsilon \mu'}$
(EASSIGNP1)	$\frac{e_1 \mu \hookrightarrow_n \varepsilon \mu'}{e_1 \leftarrow e_2 \mu \hookrightarrow_{n+1} \varepsilon \mu'}$
(EASSIGNP2)	$\frac{e_1 \mu_1 \hookrightarrow_n l \mu_2 \quad e_2 \mu_2 \hookrightarrow_n \varepsilon \mu_3}{e_1 \leftarrow e_2 \mu_1 \hookrightarrow_{n+1} \varepsilon \mu_3}$
(EDEREFT)	$\frac{e \mu \hookrightarrow_n v \mu' \quad \nexists l. v = l}{!e \mu \hookrightarrow_{n+1} \text{TypeError} \mu'}$
(EASSIGNT)	$\frac{e_1 \mu \hookrightarrow_n v \mu' \quad \nexists l. v = l}{e_1 \leftarrow e_2 \mu \hookrightarrow_{n+1} \text{TypeError} \mu'}$

$((\lambda(z : ?) \langle \text{number} \rangle ((\lambda(x : \text{number}) (\text{succ } x)) \langle \text{number} \rangle z)) \langle ? \rangle 1)$

and then another function application gives us

 $\langle \text{number} \rangle ((\lambda(x : \text{number}) (\text{succ } x)) \langle \text{number} \rangle \langle ? \rangle 1)$

We then apply the cast rule for ground types (ECSTG).

 $\langle \text{number} \rangle ((\lambda(x : \text{number}) (\text{succ } x)) 1)$

followed by another function application:

 $\langle \text{number} \rangle (\text{succ } 1)$

Then by (EDELTA) we have

 $\langle \text{number} \rangle 2$

and by (ECSTG) we finally have the result

2

5.4 Type Safety

Towards proving type safety we prove the usual lemmas. First, environment expansion and contraction does not change typing derivations. Also, changing the store typing environment does not change the typing derivations as long as the new store typing agrees with the old one. The function *Vars* returns the free and bound variables of an expression.

Lemma 6 (Environment Expansion and Contraction).

- If $\Gamma | \Sigma \vdash e : \tau$ and $x \notin \text{Vars } e$ then $\Gamma(x \mapsto \sigma) | \Sigma \vdash e : \tau$.
- If $\Gamma(y \mapsto \nu) | \Sigma \vdash e : \tau$ and $y \notin \text{Vars } e$ then $\Gamma | \Sigma \vdash e : \tau$.
- If $\Gamma | \Sigma \vdash e : \tau$ and $\bigwedge l$. If $l \in \text{dom } \Sigma$ then $\Sigma' l = \Sigma l$. then $\Gamma | \Sigma' \vdash e : \tau$.

Proof Sketch. These properties are proved by induction on the typing derivation. \square

Also, substitution does not change the type of an expression.

Lemma 7 (Substitution preserves typing). If $\Gamma(x \mapsto \sigma) | \Sigma \vdash e : \tau$ and $\Gamma | \Sigma \vdash e' : \sigma$ then $\Gamma | \Sigma \vdash [x := e']e : \tau$.

Proof Sketch. The proof is by strong induction on the size of the expression e , using the inversion and environment expansion lemmas. \square

Definition 1. The store typing judgment, written $\Gamma | \Sigma \models \mu$, holds when the domains of Σ and μ are equal and when for every location l in the domain of Σ there exists a type τ such that $\Gamma | \Sigma \vdash \mu(l) : \tau$.

Next we prove that n -depth evaluation for the intermediate language $\lambda_{\rightarrow}^{(\tau)}$ is sound. Informally, this lemma says that evaluation produces either a value of the appropriate type, a cast error, or *KillError* (because evaluation is cut short), but never a type error. The placement of $e | \mu \hookrightarrow_n r | \mu'$ in the conclusion of the lemma proves that our evaluation rules are complete, analogous to a progress lemma for small-step semantics. This placement would normally be a naive mistake because not all programs terminate. However, by using n -depth evaluation, we can construct a judgment regardless of whether the program is non-terminating because evaluation is always cut short if the derivation depth exceeds n . But does this lemma handle all terminating programs? The lemma is (implicitly) universally quantified over the evaluation depth n . For every program that terminates there is a depth that will allow it to terminate, and this lemma will hold for that depth. Thus, this lemma applies to all terminating programs and does not apply to

non-terminating program, as we intend. We learned of this technique from Ernst, Ostermann, and Cook [11], but its origins go back at least to Volpano and Smith [41].

Lemma 8 (Soundness of evaluation). If $\emptyset | \Sigma \vdash e : \tau \wedge \emptyset | \Sigma \models \mu$ then $\exists r \mu' \Sigma'. e | \mu \hookrightarrow_n r | \mu' \wedge \emptyset | \Sigma' \models \mu' \wedge (\forall l. l \in \text{dom } \Sigma \longrightarrow \Sigma' l = \Sigma l) \wedge ((\exists v. r = v \wedge v \in \mathbb{V} \wedge \emptyset | \Sigma' \vdash v : \tau) \vee r = \text{CastError} \vee r = \text{KillError})$.

Proof. The proof is by strong induction on the evaluation depth. We then perform case analysis on the final step of the typing judgment. The case for function application uses the substitution lemma and the case for casts uses environment expansion. The cases for references and assign use the lemma for changing the store typing. The inversion lemmas are used throughout. \square

Theorem 2 (Type safety). If $\emptyset \vdash e \Rightarrow e' : \tau$ then $\exists r \mu \Sigma. e' | \emptyset \hookrightarrow_n r | \mu \wedge ((\exists v. r = v \wedge v \in \mathbb{V} \wedge \emptyset | \Sigma \vdash v : \tau) \vee r = \text{CastError} \vee r = \text{KillError})$.

Proof. Apply Lemma 3 and then Lemma 8. \square

6. Relation to Dynamic of Abadi et al.

We defined the semantics for $\lambda_{\rightarrow}^?$ with a translation to $\lambda_{\rightarrow}^{(\tau)}$, a language with explicit casts. Perhaps a more obvious choice for intermediate language would be the pre-existing language of explicit casts of Abadi et. al [1]. However, there does not seem to be a straightforward translation from $\lambda_{\rightarrow}^{(\tau)}$ to their language. Consider the evaluation rule (ECSTF) and how that functionality might be implemented in terms of typecase. The parameter z must be cast to τ , which is not known statically but only dynamically. To implement this cast we would need to dispatch based on τ , perhaps with a typecase. However, typecase must be applied to a value, and there is no way for us to obtain a value of type τ from a value of type $\tau \rightarrow \tau'$. Quoting from [1]:

Neither *tostring* nor *typetostring* quite does its job: for example, when *tostring* gets to a function, it stops without giving any more information about the function. It can do no better, given the mechanisms we have described, since there is no effective way to get from a function value to an element of its domain or codomain.

Of course, if their language were to be extended with a construct for performing case analysis on types, such as the *typerec* of Harper and Morrisett [19], it would be straightforward to implement the appropriate casting behavior.

7. Related Work

Several programming languages provide gradual typing to some degree, such as Cecil [8], Boo [10], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [26], extensions to Java proposed by Gray et al. [17], and the Bigloo [6, 36] dialect of Scheme [24]. This paper formalizes a type system that provides a theoretical foundation for these languages.

Common LISP [23] and Dylan [12, 37] include optional type annotations, but the annotations are not used for type checking, they are used to improve performance.

Cartwright and Fagan's Soft Typing [7] improves the performance of dynamically typed languages by inferring types and removing the associated run-time dispatching. They do not focus on statically

catching type errors, as we do here, and do not study a source language with optional type annotations.

Anderson and Drossopoulou formalize BabyJ [2], an object-oriented language inspired by JavaScript. BabyJ has a nominal type system, so types are class names and the permissive type $*$. In the type rules for BabyJ, whenever equality on types would normally be used, they instead use the relation $\tau_1 \approx \tau_2$ which holds whenever τ_1 and τ_2 are the same name, or when at least one of them is the permissive type $*$. Our unknown type $?$ is similar to the permissive type $*$, however, the setting of our work is a structural type system and our type compatibility relation \sim takes into account function types.

Riely and Hennessy [35] define a partial type system for $D\pi$, a distributed π -calculus. Their system allows some locations to be untyped and assigns such locations the type !bad . Their type system, like Quasi-Static Typing, relies on subtyping, however they treat !bad as “bottom”, which allows objects of type !bad to be implicitly coercible to any other type.

Gradual typing is syntactically similar to type inferencing [9, 21, 27]: both approaches allow type annotations to be omitted. However, with type inference, the type system tries to reconstruct what the type annotations should be, and if it cannot, rejects the program. In contrast, a gradual type system accepts that it does not know certain types and inserts run-time casts.

Henglein [20] presents a translation from untyped λ -terms to a coercion calculus with explicit casts. These casts make explicit the tagging, untagging, and tag-checking operations that occur during the execution of a language with latent (dynamic) typing. Henglein’s coercion calculus seems to be closely related to our $\lambda^{\langle ? \rangle}$ but we have not yet formalized the relation. Henglein does not study a source language with partially typed terms with a static type system, as we do here. Instead, his source language is a dynamically typed language.

Bracha [4] defines *optional type systems* as type systems that do not affect the semantics of the language and where type annotations are optional. Bracha cites Strongtalk [5] as an example of an optional type system, however, that work does not define a formal type system or describe how omitted type annotations are treated.

Ou et. all. [31] define a language that combines standard static typing with more powerful dependent typing. Implicit coercions are allowed to and from dependent types and run-time checks are inserted. This combination of a weaker and a stronger type system is analogous to the combination of dynamic typing and static typing presented in this paper.

Flanagan [15] introduces Hybrid Type Checking, which combines standard static typing with refinement types, where the refinements may express arbitrary predicates. The type system tries to satisfy the predicates using automated theorem proving, but when no conclusive answer is given, the system inserts run-time checks. This work is also analogous to ours in that it combines a weaker and stronger type system, allowing implicit coercions between the two systems and inserting run-time checks. One notable difference between our system and Flanagan’s is that his is based on subtyping whereas ours is based on the consistency relation.

Gronski, Knowles, Tomb, Freund, and Flanagan [18] developed the Sage language which provides Hybrid Type Checking and also a Dynamic type with implicit (run-time checked) down-casts. Surprisingly, the Sage type system does not allow implicit down-casts from Dynamic, whereas the Sage type checking (and compilation) algorithm does allow implicit down-casts. It may be that the given type system was intended to characterize the output of compilation (though it is missing a rule for cast), but then a type system for the source language remains to be defined. The Sage technical re-

port [18] does not include a result such as Theorem 1 of this paper to show that the type system catches all type errors for fully annotated programs, which is a tricky property to achieve in the presence of a Dynamic type with implicit down-casts.

There are many interesting issues regarding efficient representations for values in a language that mixes static and dynamic typing. The issues are the same as for parametric polymorphism (dynamic typing is just a different kind of polymorphism). Leroy [25] discusses the use of mixing boxed and unboxed representations and such an approach is also possible for our gradual type system. Shao [38] further improves on Leroy’s mixed approach by showing how it can be combined with the type-passing approach of Harper and Morrisett [19] and thereby provide support for recursive and mutable types.

8. Conclusion

The debate between dynamic and static typing has continued for several decades, with good reason. There are convincing arguments for both sides. Dynamic typing is better suited than static for prototyping, scripting, and gluing components, whereas static typing is better suited for algorithms, data-structures, and systems programming. It is common practice for programmers to start development of a program in a dynamic language and then translate to a static language midway through development. However, static and dynamic languages are often radically different, making this translation difficult and error prone. Ideally, migrating between dynamic to static could take place gradually and while staying within the same language.

In this paper we present the formal definition of the language $\lambda^{\langle ? \rangle}$, including its static and dynamic semantics. This language captures the key ingredients for implementing gradual typing in functional languages. The language $\lambda^{\langle ? \rangle}$ provides the flexibility of dynamically typed languages when type annotations are omitted by the programmer and provides the benefits of static checking when all function parameters are annotated, including the safety guarantees (Theorem 1) and the time and space efficiency (Lemma 5). Furthermore, the cost of dynamism is “pay-as-you-go”, so partially annotated programs enjoy the benefits of static typing to the degree that they are annotated. We prove type safety for $\lambda^{\langle ? \rangle}$ (Theorem 2); the type system prevents type violations from occurring at run-time, either by catching the errors statically or by catching them dynamically with a cast exception. The type system and run-time semantics of $\lambda^{\langle ? \rangle}$ is relatively straightforward, so it is suitable for practical languages.

As future work, we intend to investigate the interaction between our gradual type system and types such as lists, arrays, algebraic data types, and implicit coercions between types, such as the types in Scheme’s numerical tower. We also plan to investigate the interaction between gradual typing and parametric polymorphism [16, 34] and Hindley-Milner inference [9, 21, 27]. We have implemented and tested an interpreter for the $\lambda^{\langle ? \rangle}$ calculus. As future work we intend to incorporate gradual typing as presented here into a mainstream dynamically typed programming language and perform studies to evaluate whether gradual typing can benefit programmer productivity.

Acknowledgments

We thank the anonymous reviewers for their suggestions. We thank Emir Pasalic the members of the Resource Aware Programming Laboratory for reading drafts and suggesting improvements. This work was supported by NSF ITR-0113569 Putting Multi-Stage Annotations to Work, Texas ATP 003604-0032-2003 Advanced

Languages Techniques for Device Drivers, and NSF SOD-0439017
Synthesizing Device Drivers.

References

- [1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
- [2] C. Anderson and S. Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82. Elsevier, 2003.
- [3] H. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic*. Elsevier, 1984.
- [4] G. Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.
- [5] G. Bracha and D. Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.
- [6] Y. Bres, B. P. Serpette, and M. Serrano. Compiling scheme programs to .NET common intermediate language. In *2nd International Workshop on .NET Technologies*, Pilzen, Czech Republic, May 2004.
- [7] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.
- [8] C. Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.
- [9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.
- [10] R. B. de Oliveira. The Boo programming language. <http://boo.codehaus.org>, 2005.
- [11] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.
- [12] N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.
- [13] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming*, October 2002.
- [14] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming*, 2004.
- [15] C. Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.
- [16] J.-Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, Paris, France, 1972.
- [17] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.
- [18] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. Technical report, University of California, Santa Cruz, 2006.
- [19] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, New York, NY, USA, 1995. ACM Press.
- [20] F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.
- [21] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans AMS*, 146:29–60, 1969.
- [22] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
- [23] G. L. S. Jr. An overview of COMMON LISP. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107, New York, NY, USA, 1982. ACM Press.
- [24] R. Kelsey, W. Clinger, and J. R. (eds.). Revised⁵ report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.
- [25] X. Leroy. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188, New York, NY, USA, 1992. ACM Press.
- [26] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.
- [27] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
- [28] T. Nipkow. Structured proofs in Isar/HOL. In *TYPES*, number 2646 in LNCS, 2002.
- [29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS. Springer, 2002.
- [30] A. Oliart. An algorithm for inferring quasi-static types. Technical Report 1994-013, Boston University, 1994.
- [31] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types (extended abstract). In *3rd IFIP International Conference on Theoretical Computer Science*, August 2004.
- [32] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
- [33] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

- [34] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).
- [35] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–104, New York, NY, USA, 1999. ACM Press.
- [36] M. Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.
- [37] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.
- [38] Z. Shao. Flexible representation analysis. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 85–98, New York, NY, USA, 1997. ACM Press.
- [39] J. Siek and W. Taha. Gradual typing: Isabelle/isar formalization. Technical Report TR06-874, Rice University, Houston, Texas, 2006.
- [40] S. Thatte. Quasi-static typing. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381, New York, NY, USA, 1990. ACM Press.
- [41] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW'97: 10th Computer Security Foundations Workshop*, volume 00, page 156, Los Alamitos, CA, USA, 1997. IEEE Computer Society.
- [42] M. Wenzel. *The Isabelle/Isar Reference Manual*. TU München, April 2004.

SOFT TYPING

Robert Cartwright and Mike Fagan
Department of Computer Science
Rice University
Houston, TX 77251-1892

Summary

Type systems are designed to prevent the improper use of program operations. They can be classified as either *static* or *dynamic*. Static type systems detect “ill-typed” programs at compile-time and prevent their execution. Dynamic type systems detect “ill-typed” programs at run-time.

Static type systems have two important advantages over dynamic type systems. First, they provide important feedback to the programmer by detecting a large class of program errors *before* execution. Second, they extract information that a compiler can exploit to produce more efficient code. The price paid for these advantages, however, is a loss of expressiveness, modularity, and semantic simplicity.

This paper presents a *soft* type systems that retains the expressiveness of dynamic typing, but offers the early error detection and improved optimization capabilities of static typing. The key idea underlying soft typing is that a type checker need not *reject* programs containing “ill-typed” phrases. Instead, the type checker can insert explicit run-time checks, transforming “ill-typed” programs into type-correct ones.

1 Introduction

Computations in high-level programming languages are expressed in terms of operations on abstract values such as integers, matrices, sequences, trees, and functions. Most of these operations are *partial*: they are defined only for inputs of a certain form. For example, the `head` operation (`car` in LISP) is defined only for non-empty sequences.

Since programmers can write programs with undefined applications, a programming language must assign some form of meaning to undefined applications. To address this problem, most programming languages adopt a type discipline. This discipline can either be *static* or *dynamic*. A *statically typed language* prevents undefined applications by imposing syntactic restrictions on programs that *guarantee* the definedness of every application. The implementation refuses to execute programs that do not meet these restrictions. ML[17] is a prominent example of a contemporary statically typed language.

In contrast, a *dynamically typed language* accepts all programs. Undefined applications are detected during program execution by tests embedded in the code implementing each primitive program operation. If an operation’s arguments are not acceptable, the operation generates a program exception, transferring control to an exception handler in the program or the operating system. Scheme[5] is a prominent example of a dynamically typed programming language.

Static typing has two important advantages over dynamic typing:

Error Detection: The system flags all “suspect” program phrases before execution.

Optimization: The compiler produce better code by independently optimizing the representation of each type and eliminating *most* run-time checks

However, these advantages are achieved at the cost of a loss of expressiveness, modularity, and semantic simplicity:

Expressiveness: no type checker can always decide whether or not a program will generate any undefined applications. Therefore, the type checker must err on the side of safety and reject some programs that do not contain any semantic errors. As a result, they reject many programs that lie beyond the limited deductive power of the type-checking system.

Modularity: some procedures in program libraries and some procedures exported from modules will be assigned more restrictive types than their meaning requires, restricting the contexts in which they can be used.

Semantic Simplicity: the type system (a complex set of syntactic rules) is an essential part of the language definition that a programmer must understand to write programs. Otherwise, he will repeatedly trip over the syntactic restrictions imposed by the type system.

In this paper, we present a generalization of static and dynamic typing—called *soft typing*—that combines the best features of both approaches. A soft type system can statically type check *all* programs written in a dynamically typed language because the type checker is permitted to insert explicit run-time checks around the arguments of “suspect” applications. In other words, the type checker transforms source programs to type correct programs by judiciously inserting run-time checks.

Soft typing retains the advantages of static typing because the final result is a type correct program in a sublanguage that conforms to a static type discipline. In the context of soft typing, a programmer can detect errors before program execution by inspecting each run-time check inserted by the type checker. Compilers can generate efficient code for softly typed languages because the transformed program is type correct.

The key technical obstacle to constructing a *practical* soft type system is devising a type system that is

- rich enough to type most program components written in a dynamic typing style without inserting any run-time checks, yet
- simple enough to accommodate *automatic type assignment* (often called *type inference* or *type reconstruction*).

Existing static type systems fail to satisfy the first test, namely, the typing of most program components written in a dynamic style *without inserting run-time checks*.

In this paper, we show how to construct a practical soft type system that subsumes the ML type system. Our work adapts an encoding technique developed by Rémy. This technique enables our soft type system to rely on essentially the same type assignment algorithm as ML. The only difference is that our algorithm uses circular unification instead of ordinary unification. Rémy’s encoding also enables us to use the ML type assignment algorithm on a transformed problem to determine what run-time checks to insert.

The rest of the paper is organized as follows. Section 2 presents the technical preliminaries. Section 3 identifies the appropriate design criteria for a soft type system, focusing on the necessary elements in the type language. Section 4 describes a type language that meet these criteria. Section 5 presents an inference system for deducing types for program expressions and Section 6 gives an algorithm that automatically assigns types to program expressions. Section 7 describes a method for inserting run-time checks when the type assignment algorithm fails. Finally, Section 8 discusses related work, and Section 9 summarizes our principal results.

2 Exp: a Simple Programming Language

We are interested in developing a soft type system suitable for higher order imperative languages like Scheme and Standard ML. For the sake of simplicity, we will confine our attention in this paper to the simple functional language **Exp** introduced by Milner[16] as the functional core of ML. The automatic type assignment and coercion insertion methods that we present for **Exp** can be extended to assignment and advanced control structures using the same techniques that Tofte[21], MacQueen[4], and Duba et al[9] have developed for ML.

The syntax of **Exp** is given by the following grammar:

Definition 1 (*The programming language*) Let x range over a set of variables and c range over a set of constants K

$$L ::= x \mid c \mid \lambda x . L \mid (L \ L) \mid \text{let } x = L \text{ in } L$$

The set K of constants contains *constructors* that build values, *selectors* that tear apart values, and **case** functions that conditionally combine operations. For example, the boolean constants **true** and **false** are 0-ary constructors, the list operation **cons** is a binary constructor, and the **head** and **tail** functions are selectors on non-empty lists (constructed using the **cons** operation). For each non-repeating sequence of constructor names, there is a **case** function. If $\langle c_1, \dots, c_n \rangle$ is a sequence of constructor names, then $\text{case}_{\langle c_1, \dots, c_n \rangle}$ takes $n + 1$ arguments: a value v , and n functions o_1, \dots, o_n . The $\text{case}_{\langle c_1, \dots, c_n \rangle}$ is informally defined by the following equation:

$$\text{case}_{\langle c_1, \dots, c_n \rangle}(v)(o_1) \dots (o_n) = \begin{cases} o_1(v) & v \text{ is a } c_1 \text{ construction} \\ \dots \\ o_n(v) & v \text{ is a } c_n \text{ construction} \\ \text{fatal-error} & \text{otherwise} \end{cases}$$

The standard ternary **if** operation is synonymous with the function $\text{case}_{\text{true}, \text{false}}$.

Exp is a conventional *call-by-value* functional language. With the exception of the primitive **case** functions, all functions diverge if their arguments diverge. The formal semantic definition for **Exp** is given in an Appendix in the full paper. The only unusual feature of the semantics is the inclusion of a special element **fatal-error** in the data domain to model failed applications which can never be executed in type correct programs. The data domain also includes exception values to model the output of dynamic run-time checks.

3 Criteria for a Soft Type System

From the perspective of a programmer, a softly typed language is a dynamically typed language that provides feedback on *possible* type errors. From the perspective of a compiler-writer, it is statically typed language because only type correct programs are compiled and executed.

For a soft type system to be effective and practical, it must satisfy the following two criteria:

Minimal-Text-Principle The system should not require the programmer to declare the types of any program phrases or operations.

Minimal-Failure-Principle The system should leave “most” program components unchanged unless they can produce type errors during execution.

To satisfy the minimal-failure condition, a soft type system must accommodate *parametric polymorphism*, an important form of modularity found in dynamically typed languages. Parametric polymorphism is best explained by giving an example. Consider the well-known LISP function **reverse** that takes a list as input and reverses it. For any type α , **reverse** maps the type $\text{list}(\alpha)$ to $\text{list}(\alpha)$. To propagate precise type information about a specific application of **reverse**, we need to capture the fact that the elements of the output list belong to the same type as the elements of the input elements. Otherwise, we will not be able to type check subsequent operations on the output elements. For this reason, a soft typing system must be able to express the fact that **reverse** has type $\forall \alpha. \text{list}(\alpha) \rightarrow \text{list}(\alpha)$ as in ML.

A type system based only on parametric polymorphism, however, still does not satisfy minimal-failure criterion. There are two important classes of program expressions that commonly occur in dynamically typed programs that do not type-check in languages that support only parametric polymorphism.

3.1 Non-uniformity

The first class of troublesome program expressions is the set of expressions that do not return “uniform” results. Consider the following sample function definition.

Example 1 (Non-uniform if) Let f be the function

$$\lambda x. \text{if } x \text{ then } 1 \text{ else nil}$$

where **if-then-else** has type $\forall \alpha \text{ bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$.

f takes a simple boolean value and returns either 1 or **nil**. Clearly, no run-time error occurs if x is a boolean. Nevertheless, this function fails to type check, because 1 and **nil** belong to different types.

To assign types to non-uniform expressions, we need to introduce *union* types. The union type $\alpha \cup \beta$ is the union of the sets α and β . It is a fundamentally different type construction than the *disjoint union* type construction found in many languages (including ML). The *disjoint union* of α and β forms a new set disjoint from α and β ; the elements of new set are created by “tagging” the elements of α and β .

In a type system with union types, the function in Example 1 has type $\text{bool} \rightarrow \text{int} \cup \text{nil}$. Union types also enable us to make finer grained type distinctions. As an illustration of this effect, consider the type **list**. It is the union of two different constructor types: the empty list **nil** and the constructed lists of the form $(\text{cons } \alpha L)$. The division of the type $\text{list}(\alpha)$ into the union of the types **nil** and $\text{cons}(\alpha)$ is important because the selectors **head** and **tail** functions produce run-time errors when applied to the empty list! Languages without union types typically define the type of **head** as $\text{list}(\alpha) \rightarrow \alpha$. In this case, $(\text{head } \text{nil})$ will pass the type checker, even though it will generate a run-time error when it is executed.

3.2 Recursivity

The other troublesome class of program expressions is the set of expressions that induce recursive type constraints. A classic example of this phenomenon is the self application function

$$S = \lambda x.(x \ x)$$

Since x is applied as a function in the body of S , x must have type $\alpha \rightarrow \beta$. But the self-application also forces $\alpha \rightarrow \beta$ to be subtype of the input type of x which is α . In a dynamically typed language like Scheme, the function S can be written and executed. If S is applied to the identity function or a constant function, it will produce a well-formed answer. Nearly all static type systems, however, reject S .

The simplest way to construct solutions to recursive type constraints is to include a fixed-point operator **fix** in the language of type terms. This feature is usually called *recursive typing*. Given the operator **fix**, we can assign the type $\text{fix } t.t \rightarrow \beta$ to S .

The true power of recursive typing, however, lies in its use with union types. The following simple example (taken from an introductory Scheme course[10]) is a good illustration of this phenomenon.

Example 2 Let the function `deep` be defined by the equation

$$\text{deep} = \lambda n. (\text{if } (=? n \text{ zero}) \text{ zero} (\text{cons} (\text{deep} (\text{pred} n)) \text{ nil}))$$

where `zero` is the 0-ary constructor denoting 0 and `pred : suc → zero ∪ suc` is the standard predecessor function.

This function takes a non-negative integer n as input, and returns 0 nested inside n levels of parentheses. It does not type check in ML because the output type cannot be described by a simply parametric pattern or cyclic pattern. In a type system with parametric types, union types, and recursive types, `deep` has type $(\text{zero} \cup \text{suc}) \rightarrow \text{fix } t.\text{zero} \cup \text{cons}(t)$

4 A Type Language Suitable for Soft Typing

Our type language is parameterized by the set of data constructors in the programming language. We assume that the data constructors are disjoint: no value v can be generated by two different constructors. For each data constructor c , we define a type constructor with the same name c . The type language also includes the special type constructor \rightarrow for defining function types. The arity of each type constructor depends on the degree of polymorphism inherent in the corresponding data constructor. For example, the data constructor `cons` for building lists accepts arbitrary values as its first argument, but the second argument must be a list, so the `cons` type constructor has arity 1. Similarly, the data constructor `suc` takes one argument, but it must be a non-negative integer, so the type constructor `suc` arity 0. The special type constructor \rightarrow has arity 2.

Given a set of type constructors C , we define the set of *constructor* type terms $\Sigma(C, V)$ over a set of variables V to be the free term algebra over C, V . We define the *raw* type terms over C and V as the set that is inductively defined by the equation

$$T = V \mid c(\dots) \mid T + T \mid \text{fix } v.T.$$

where v is any type variable in V , c is any type constructor in C , and \dots is a list of n raw type terms where n is the arity of c .

The intended meaning of raw type terms is the obvious one. Every closed type term denotes a set of data objects. For each data constructor $c \in C$, the corresponding type construction $c(\alpha_1, \dots, \alpha_m)$ denotes the set of all data objects of the form $c(x_1, \dots, x_n)$ where (i) each polymorphic argument x_i in the list x_1, \dots, x_n is taken from the type denoted by the matching¹ argument α_j and (ii) each non-polymorphic argument x_i is taken from the type specified in the definition of the data constructor c . The function type construction $\alpha \rightarrow \beta$ denotes the set of all continuous functions that map α into β . The $+$ operator denotes the union operation on sets and the `fix` operator denotes the least fixed-point operation

¹Type argument α_i corresponds to the i th *polymorphic* argument of the data constructor c . Hence, j may be greater than i .

on type functions. A formal definition of the semantics of type terms based on the ideal model[15, 14] of MacQueen, Plotkin, and Sethi is given in the full paper.

For technical reasons, we must impose some modest restrictions on the usage of the **fix** and + operators in type terms. The set of *tidy type terms* over C and V is the set of raw type terms over C and V that satisfy the following two constraints:

1. Every subterm of the form $\text{fix } v.t$ is *formally contractive*. Almost all uses of **fix** that arise in practice are formally contractive, but the formal definition is tedious. It is given in the full paper. The expressions $\text{fix } \alpha.\alpha$ and $\text{fix } \alpha.\alpha + \text{nil}$ are not formally contractive, but $\text{fix } \alpha.\alpha \rightarrow \beta$ and $\text{fix } \alpha.\text{nil} + \text{cons}(\alpha)$ are.
2. Every subterm of the form $u + v$ is *discriminative*. A raw type term of the form $u + v$ is *discriminative* iff (i) neither u or v is a type variable, and (ii) each type constructor appears *at most once* at the top level (not nested inside another type construction) of $u + v$. The terms $t + \text{true}$ and $\text{cons}(x) + \text{cons}(y)$ are not discriminative, but $\text{false} + \text{true}$ and $\text{nil} + \text{cons}(y)$ are.

Let C be the set of type constructors for **Exp**. Given a set of type variables V , the *soft type language* for **Exp** is the set of *tidy type terms* over C and V .

Our soft type system is specifically designed to facilitate reasoning about subtypes of unions, an essential characteristic of soft type systems that we identified in Section 3. To reason about subtypes, we use the following inference system patterned after a system developed by Amadio and Cardelli[3]

Definition 2 (*Inference Rules for Subtyping*)

$$\begin{array}{ll}
\text{REFL:} & \vdash T \subseteq T \\
\text{UNI:} & \vdash T_1 \subseteq T_1 + T_2 \\
\text{FIX1:} & \vdash \text{fix } x.T = T[x \leftarrow \text{fix } x.T] \\
\text{TRANS:} & \frac{\vdash T_1 \subseteq T_2 \quad \vdash T_2 \subseteq T_3}{\vdash T_1 \subseteq T_3} \\
\text{CON:} & \frac{\vdash S_1 \subseteq T_1 \dots \vdash S_n \subseteq T_n \quad \vdash c(S_1, \dots, S_n) \subseteq c(T_1, \dots, T_n)}{c \text{ an } n\text{-ary constructor, } c \neq \rightarrow} \\
\text{FUN:} & \frac{\vdash T_3 \subseteq T_1 \quad \vdash T_2 \subseteq T_4}{\vdash T_1 \rightarrow T_2 \subseteq T_3 \rightarrow T_4} \\
\text{FIX2:} & \frac{t_1 \subseteq t_2 \vdash T_1[x \leftarrow t_1] \subseteq T_2[x \leftarrow t_2]}{\vdash \text{fix } x.T_1 \subseteq \text{fix } x.T_2}
\end{array}$$

The inference rules assume that + operator is associative, commutative, and idempotent.

The final step in defining our type language is adding the notion of universal quantification. As in ML[8, 7], we restrict universal quantification to the “outside” of type terms. In this restricted setting, type terms containing quantifiers are called *type schemes*. A type scheme is defined by the grammar:

$$\sigma ::= \tau | \forall t. \sigma$$

where τ denotes the set of tidy terms. The types of polymorphic operations are given by type schemes rather than type terms.

The semantics for tidy type terms is easily extended to accommodate type schemes. The technical details appear in the full paper.

5 Type Inference

To assign types to programming language expressions, we use a type inference system similar to the type inference system for ML. The inference system uses both the subtype relation \subseteq defined by the subtype inference system given in Section 4 and the *generic instance* relation \leq defined by Damas and Milner[7] for the ML type system. A generic instance of type scheme τ is a type scheme τ' obtained by substituting tidy type terms for *quantified* variables of τ . For example, given the type $\forall\alpha.\text{cons}(\alpha) \rightarrow \alpha$, then $\text{cons}(\text{int}) \rightarrow \text{int}$ is a generic instance.

The inference rules below are the original Milner rules augmented by the single rule SUB

Definition 3 (*The soft type system rules*) Let e be a program expression and let t be a type scheme. The formula $e : t$ is called a *typing judgment*. The intended meaning is that e has type t .

Let A be a set of type assumptions of the form $\text{id} : \sigma$.

$$\begin{array}{lll}
 \text{TAUT:} & A \vdash x : \sigma & A(x) = \sigma \\
 \text{INST:} & \frac{A \vdash x : \sigma}{A \vdash x : \sigma'} & \sigma' \leq \sigma \\
 \text{GEN:} & \frac{A \vdash e : \sigma}{A \vdash e : \forall\alpha\sigma} & \alpha \text{ not free in } A \\
 \text{ABST:} & \frac{A \cup \{x : \tau'\} \vdash e_1 : \tau}{A \vdash \lambda x.e_1 : \tau' \rightarrow \tau} \\
 \text{APP:} & \frac{A \vdash f : \tau_1 \rightarrow \tau_2 \quad A \vdash e : \tau_1}{A \vdash (f e) : \tau_2} \\
 \text{LET:} & \frac{A \vdash e_1 : \sigma \quad A \cup \{x : \sigma\} \vdash e_2 : \tau}{A \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \\
 \text{SUB:} & \frac{A \vdash e : \tau}{A \vdash e : \tau'} & \tau \subseteq \tau'
 \end{array}$$

The effect of adding the SUB rule is surprisingly subtle. The most important consequence is that program expressions do not necessarily have best (*principal*) types. The following counterexample demonstrates this fact.

Example 3 (*A non uniform deduction*) Let $f_1 : (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ and $f_2 : a + b \rightarrow a$ be program functions. We can immediately deduce that $(f_1 f_2) : a \rightarrow a$. Alternatively, $A \vdash (f_1 f_2) : a + b \rightarrow a + b$. These two typings for $(f_1 f_2)$ are incomparable, so neither is best. Furthermore, no other type that has both types as supertypes can be derived, so there exists no best type.

Nevertheless, our type system retains the most important property of the ML type inference system, namely the soundness property and its corollaries. The soundness property simply asserts that every provable typing judgment is true, according to semantics that we have defined for the programming language, the type language, and type assertion language. This result is formally stated in the full paper. On an informal level, the soundness theorem ensures that well-typed programs never execute undefined function applications.

6 Automated Type Assignment

To satisfy the minimal-text criterion for soft typing, we must produce a practical algorithm for assigning types to program expressions. We have developed an algorithm based on the type assignment algorithm from ML (Milner’s algorithm W). Our algorithm differs from the ML algorithm in two respects.

- First, we encode all program types as instances of a single polymorphic type that has type parameters for every type constructor and for every parameter to a type constructor. This encoding was developed by Rémy[19] to reduce inheritance polymorphism to parametric polymorphism.
- Second, we use circular unification (as in CAML[22]) instead of regular unification to solve systems of type equations. Circular unification is required to infer recursive types.

The Rémy encoding is based on the observation that set of data constructors C is *finite*. Consequently, we can enumerate (up to substitution instances) all the tidy subtypes and supertypes of a given type expression. Consider the following example. Assume that the only constructors in our type language are $a(x), b, c$. Then the supertypes of the type b are $\{b, (a(x)+b, b+c, a(x)+b+c)\}$. We can describe this set using a simple pattern by analyzing which type constructors *must* appear in a supertype, which type constructors *may* appear in a supertype, and which type constructors *must not* appear in a supertype.

The following table presents the results of this analysis:

Status a	Status b	Status c
may	must	may

The same form of analysis can be used to construct a pattern describing all the possible subtypes of a given type. Since b has no subtypes other than itself, let’s consider the type $b + c$ instead. The subtypes of $b + c$ are $\{b, c, b + c\}$. A “must/may/must-not” analysis of this set yields the table

Status a	Status b	Status c
must-not	may	may

Note that in supertype patterns, positive (“must”) information plays a crucial role, while in subtype patterns, negative (“must-not”) information is crucial. We need both forms of patterns because the function type constructor \rightarrow inverts the form of information provided by its first argument.

Rémy expresses these patterns in the framework of parametric polymorphism by using a single highly polymorphic type constructor \mathcal{R} and two type constants $+$ and $-$ signifying “must” and “must-not” respectively. \mathcal{R} has a type parameter for each type constructor (which must be instantiated by either $+$ or $-$) and a type parameter for each type argument taken by a constructor (which must be instantiated by an instance of \mathcal{R}). Using this notation, we can rewrite the preceding patterns as follows. The supertypes b are given by the Rémy type expression $\mathcal{R}(\tau_1, x, +, \tau_2)$. The subtypes of $b + c$ are given by $\mathcal{R}(-, x, \tau_3, \tau_4)$

The key property of this encoding is that it *eliminates* the union operator $+$ and reduces the subtyping relation to the instantiation of the polymorphic type constructor \mathcal{R} . As a result, the ML type assignment algorithm can be used to solve type equations. If we extend Rémy’s notation to include the **fix** operator, then we can express all tidy types using Rémy’s notation. More precisely, we can map any tidy type term τ to a Rémy type term $R_+(\tau)$ denoting all of the supertypes of τ . Similarly, we map any tidy type term τ to a Rémy type term $R_-(\tau)$ denoting all of the subtypes of τ . The encoding is non-trivial because we must treat the function type constructor \rightarrow as a special case to cope with the fact \rightarrow is anti-monotonic in its first argument.

Before we define the mapping from tidy type terms to Rémy notation, we need to introduce some subsidiary definitions. Let \parallel denote the *syntactic concatenation operator* defined by the equation:

$$\langle a_1, \dots, a_k \rangle \parallel \langle b_1, \dots, b_l \rangle \equiv \langle a_1, \dots, a_k, b_1, \dots, b_l \rangle$$

A tidy type term τ of the form $c(t_1, \dots, t_m), t$ is a *component* of a tidy type term $u + v$ (written $\tau \subseteq u + v$) iff τ is identical to either u or v , or τ is a component of either u or v . If no term of the form $c(t_1, \dots, t_m), t$ is a component of a term w , we say that c *does not occur* in w (written $c \not\subseteq w$).

We define the mappings R_+ and R_- from tidy type terms to Rémy type terms as follows:

$$R_{+,i}(t) = \begin{cases} t & t \in V \\ \langle +, R_+(t_1), \dots, R_+(t_m) \rangle & c_i(t_1, \dots, t_m) \subseteq t \\ & c_i \neq \rightarrow \\ \langle +, R_-(t_1), \dots, R_-(t_m) \rangle & t_1 \rightarrow t_2 \subseteq t \\ \langle \kappa_j, x'_1, \dots, x'_m \rangle & c_i \not\subseteq t \\ \mathbf{fix} \ m.R_{+,i}(t') & t = \mathbf{fix} \ m.t' \end{cases}$$

$$R_{-,i}(t) = \begin{cases} t & t \in V \\ \langle \kappa_j, R_-(t_1), \dots, R_-(t_m) \rangle & c_i(t_1, \dots, t_m) \subseteq t \\ & c_i \neq \rightarrow \\ \langle \kappa_j, R_+(t_1), R_-(t_2) \rangle & t_1 \rightarrow t_m \subseteq t \\ \langle -, x'_1, \dots, x'_m \rangle & c_i \not\subseteq t \\ \mathbf{fix} \ m.R_{-,i}(t') & t = \mathbf{fix} \ m.t' \end{cases}$$

$$R_+(t) = \mathcal{R}(R_{+,1}(t) \parallel R_{+,2}(t) \dots \parallel R_{+,n}(t))$$

$$R_-(t) = \mathcal{R}(R_{-,1}(t) \parallel R_{-,2}(t) \dots \parallel R_{-,n}(t))$$

where each occurrence of κ_j denotes a fresh type variable, distinct from all other type variables.

The inverse transformations are similar and will not be shown here. However, we note that some \mathcal{R} -terms correspond to a *set* of tidy terms, rather than a single term.

Our type assignment algorithm consists of three steps:

1. Encode the types of primitive operations in Rémy notation.
2. Perform ordinary type assignment via algorithm W (using circular unification).
3. Translate the Rémy type expressions back into tidy terms.

In the full paper, we will prove that our type assignment algorithm is sound.

6.1 Some examples

Consider the function `mixed = λ x.if x then 1 else nil` defined in Example 1. Our algorithm assigns the type `true + false → suc + nil` to `mixed`.

A more interesting example is the function `taut` which determines whether an arbitrary Boolean function (of any arity!) is a tautology (`true` for all inputs).

Example 4 (*Tautology example*)

```
taut = λB.case B of
    true :  true
    false:  false
    fn : ((and (taut (B true))) (taut (B false)))
```

For the `taut` function, our algorithm produces the typing `taut : β → (true + false)` where $\beta = \text{fix } t.(\text{true} + \text{false} + ((\text{true} + \text{false}) \rightarrow t))$.

7 Inserting run-time checks

As we explained in the introduction, *no* sound type checker can pass all “good” programs. Our type checker described in section 6 succeeds in inferring types for most programs. Nevertheless, some “good” programs will not pass that type checker. For example:

Example 5 (*Function with no type*)

$$N_1 = \lambda f. \text{if } f(\text{true}) \text{ then } f(5) + f(7) \text{ else } f(7)$$

The type checker fails because it infers incompatible constraints for the type of the argument f . The `if` test forces $f : \text{true} \rightarrow \text{true} + \text{false}$. Similarly, the `true` arm of the `if` requires $f : \text{suc} \rightarrow \text{z} + \text{suc}$. There is no unifier for these two typings of f , preventing the type checker from assigning a type to N_1 . The function N_1 is not badly defined, however, because $N_1(\lambda x.x)$ never goes wrong.

Sometimes our type checker fails to account for all the possible uses of a function. Consider the following modification of the previous example.

Example 6 (*An anomaly*)

$$N_2 = \lambda f. \text{if } f(\text{true}) \text{ then } f(5) \text{ else } f(7)$$

In this case, our type assignment algorithm yields the typing $N_2 : (\text{true} + \text{suc} \rightarrow \text{true} + \text{false}) \rightarrow \text{true} + \text{false}$. But the application $N_2(\lambda x.x)$ is also well defined, but does not type check.

In both examples, the program is meaningful, but the type analysis is not sufficiently powerful to assign an appropriate type. A static type system rejects these programs, in spite of their semantic content. A soft type system, however, *cannot* reject programs. It must insert explicit run-time checks instead.

To support the automatic insertion of explicit run-time checks, we force the programming language to include a collection of *exceptional values* in the data domain and a collection of functional constants to the programming language called *narrowers*. The narrowers perform run-time checks and exceptional values propagate the information that a run-time check failed. To define these notions more precisely, we need to introduce the concepts of *primitive* and *simple* types. Each type constructor determines a *primitive* type: it consists of all values that belong to some instance of the type. We denote the primitive type corresponding to a constructor c by the name of the constructor c . Since the type constructors are disjoint, every value (other than errors) belongs to *exactly one* primitive type. For values that are not functions, the primitive type of the value is simply the outermost constructor in the representation of the value. For a function, the primitive type is simply \rightarrow . A *simple* type is simply a union of primitive types. Using notation analogous to type terms, we denote the simple type consisting of the union of primitive types t_1, \dots, t_n by the expression $t_1 + \dots + t_n$.

There is one exceptional value for each simple type S ; it is denoted error_S . Similarly, there is a narrower \downarrow_T^S for every pair of simple types such that $S \subseteq T$. It is defined by the equation

$$\downarrow_T^S(v) = \begin{cases} v & v \in T \\ \text{error}_T & v \in S - T \\ \text{fatal-error} & \text{otherwise} \end{cases}$$

The type associated with the narrower \downarrow_T^S is $s_1 + \dots \rightarrow t_1 + \dots$ where $S = \{s_1, \dots\}$ and $T = \{t_1, \dots\}$.

The type insertion algorithm consists of the following sequence of steps:

1. Encode all type assumptions about primitive operations using R_+ and convert all “ $-$ ” flags to fresh type variables—eliminating all negative information.
2. Apply algorithm W (with circular unification!) on this transformed program to construct a “positive” type for every program expression.
3. For each occurrence of a primitive, unify the “positive” type with original type (using Rémy encodings). If unification fails, insert the narrower required for unification.

In the full paper, we prove that the narrower insertion algorithm preserves the meaning of programs and that it always succeeds in constructing a type correct program. Both proofs are straightforward.

7.1 Examples

To illustrate the insertion process, we analyze how it handles the two troublesome examples that we presented at the beginning of this section. In both of these examples, let c_1 and c_2 denote the narrowers $\downarrow_{z+suc}^{z+suc+true+false}$ and $\downarrow_{true+false}^{z+suc+true+false}$, respectively.

The insertion algorithm changes the function N_1 in example 5 to

$$N'_1 = \lambda f. \text{ if } c_1(f(\text{true})) \text{ then } c_1(f(5)) + c_1(f(7)) \text{ else } f(7)$$

The type for N'_1 is now $(z + suc + true + false \rightarrow z + suc + true + false) \rightarrow z + suc$ and the application $N'_1(\lambda x.x)$ is now type correct.

Similarly, in example 6, assume that a program contains definition of N_2 and the application $(N_2(\lambda x.x))$. Then our insertion algorithm will modify the definition of N_2 to produce

$$N'_2 = \lambda f. \text{ if } c_1(f(\text{true})) \text{ then } f(5) \text{ else } f(7)$$

Now, the type system infers the type for N'_2 as:

$$(z + suc + true + false \rightarrow z + suc + true + false) \rightarrow z + suc + true + false$$

and $N'_2(\lambda x.x)$ now type checks.

8 Related Work

The earliest work on combining static and dynamic was an investigation of the type “dynamic” conducted by Abadi et al[1]. They added dynamic data values to a conventionally statically typed data domain and provided facilities for converting data values to dynamic values by performing explicit “tagging” operations. This system permits programs written in statically typed languages to manipulate dynamic forms of data. It does not meet the criteria for soft typing because the programmer must annotate his program with explicit tagging operations to create “dynamic” values. In addition, programs are still rejected; narrowers are not inserted by the type checker.

More recently, Thatte[20] has developed the notion of “quasi-static typing” that augments a static type system with a general type Ω . Thatte’s system resembles soft typing in that it insert narrowers when necessary, ensuring that all programs can be executed. However, it does not meet the other criteria required for soft typing. It requires explicit declarations of the argument types for functions, yet cannot type check many dynamically typed programs (without inserting narrowers) because it does not support parametric polymorphism, recursive types, or more than one level of subtyping.

Researchers in the area of optimizing compilers have developed static type systems for dynamically typed languages to extract information for the purposes of code optimization. Two of the most prominent examples are the systems developed by Aiken and Murphy[2] and Cohagan and Gateley[6]. These systems, however, were not designed to be used or understood by programmers. Consequently, they lack the uniformity and generality required for soft typing.

Finally, many researchers in the area of static type systems have developed extensions to the ML type system that can type a larger fraction of “good” programs. In the arena,

the work of Mitchell[18], Fuh-Mishra[11, 12, 13], and, of course Rémy[19] is particularly noteworthy and has exerted a strong influence on our work.

9 Conclusions

The principal contribution of this research is the introduction of a new paradigm for program typing called *soft typing* that combines the best features of *static* and *dynamic* typing. The principal technical contributions include

- A type system specifically designed to assign types to a large fraction of programs written in dynamically typed languages. The type system incorporates union types, recursive types, and parametric polymorphism. The type system includes a sound type inference system for deducing types for program expressions.
- A simple yet powerful algorithm for automatically assigning types to program expressions, without the need for explicit type declarations.
- A simple algorithm for frugally inserting explicit run-time checks in programs for which the type assignment algorithm fails. Consequently, *any* program may be safely executed.

References

- [1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. In *Proceedings of the Sixteenth POPL Symposium*, 1989.
- [2] Alexander Aiken and Brian Murphy. Static type inference in a dynamically typed language. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages*, 1991. To appear.
- [3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, August 1990. (To Appear).
- [4] Andrew W. Appel and David MacQueen. Standard ML of new jersey reference manual. (in preparation), 1990.
- [5] William Clinger and Jonathan Rees. *Revised^{3.99} Report on the Algorithmic Language Scheme*, August 1990.
- [6] William Cohen and John Gateley. Interprocedural dataflow type inference. Submitted to SIGPLAN '91, 1990.
- [7] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982.
- [8] Luis Manuel Martins Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

- [9] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991. (To Appear).
- [10] D. P. Friedman and M. Felleisen. *The Little Lisper*. Science Research Associates, third edition, 1989.
- [11] You-Chin Fuh. *Design and Implementation of a Functional Language with Subtypes*. PhD thesis, State University of New York at Stony Brook, 1989.
- [12] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *Conference Record of the European Symposium on Programming*, 1988.
- [13] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT*, 1989.
- [14] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1983.
- [15] D. B. MacQueen and Ravi Sethi. A semantic model of types for applicative languages. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 1982.
- [16] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.
- [17] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [18] John C. Mitchell. Coercion and type inference. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1983.
- [19] Dider Rémy. Typechecking records and variants in a natural extension of ml. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.
- [20] Sattish Thatte. Quasi-static typing. In *Proceedings of the Seventeenth POPL Symposium*, 1990.
- [21] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1987.
- [22] Pierre Weis. *The CAML Reference Manual*. INRIA, 1987.

Linear types can change the world!

Philip Wadler
University of Glasgow*

Abstract

The linear logic of J.-Y. Girard suggests a new type system for functional languages, one which supports operations that “change the world”. Values belonging to a linear type must be used exactly once: like the world, they cannot be duplicated or destroyed. Such values require no reference counting or garbage collection, and safely admit destructive array update. Linear types extend Schmidt’s notion of single threading; provide an alternative to Hudak and Bloss’ update analysis; and offer a practical complement to Lafont and Holmström’s elegant linear languages.

An old canard against functional languages is that they cannot change the world: they do not “naturally” cope with changes of state, such as altering a location in memory, changing a pixel on a display, or sensing when a key is pressed.

As a prototypical example of this, consider the world as an array. An array (of type Arr) is a mapping from indices (of type Ix) to values (of type Val). For instance, the world might be a mapping of variable names to values, or file names to contents. At any time, we can do one of two things to the world: find the value associated with an index, or update an index to be associated with a new value.

Of course it is possible to model this functionally; we just use the two operations

$$\begin{aligned} \text{lookup} &: \text{Ix} \rightarrow \text{Arr} \rightarrow \text{Val}, \\ \text{update} &: \text{Ix} \rightarrow \text{Val} \rightarrow \text{Arr} \rightarrow \text{Arr}. \end{aligned}$$

A program that interacts with the world might have the form

$$\text{main} : \text{Args} \rightarrow \text{Arr} \rightarrow \text{Arr},$$

where the first parameter is the list of arguments that make up the command line, the second parameter is the old world, and the result is the new world. An example of a program is

$$\text{main files } a = \text{update "stdout"} (\text{concat} [\text{lookup } i a \mid i \leftarrow \text{files}]) a.$$

* Author’s address: Department of Computing Science, University of Glasgow, G12 8QQ, Scotland.
Electronic mail: wadler@cs.glasgow.ac.uk.

Presented at *IFIP TC 2 Working Conference on Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990; published in M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*, North Holland, 1990.

This performs the same operation as “cat” in Unix: it writes to the file “stdout” the result of concatenating the contents of each file named in the list *files*. (The example uses a list comprehension notation familiar from languages such as Miranda¹ [Tur85] or Haskell [HW88].)

So the canard is indeed a lie. But it is not completely without basis, because there is something unsatisfactory with this way of modelling the world. Namely, it allows operations that duplicate the world:

$$\text{copy } a = (a, a),$$

or that discard it:

$$\text{kill } a = ().$$

Neither of these correspond to our intuitive understanding of “the world”: there is one world, which (although it may change) is neither duplicated nor discarded.

This paper discusses the use of a linear type system, based on the linear logic of J.-Y. Girard [Gir87, GLT89]. Values belonging to a linear type must be used exactly once: like the world, they can be neither duplicated nor discarded.

“No duplication” helps to guarantee efficient implementation. For example, it guarantees that it is safe to update an array destructively, by overwriting the given index with the given value. Similar efficiencies can be achieved when updating a value that corresponds to a file system. If a value is represented by a linked list, then knowing that one holds the only pointer to the list may enable efficient re-use of the list cells.

“No discarding” is equally important. In most implementations of a functional language it is essential to recover space that is discarded by the use of reference counting or garbage collection. Values of a linear type avoid this overhead. As we shall see, this doesn’t mean that one cannot release storage used by a linear value; rather, it means that both allocation and deallocation of linear values are explicitly indicated in the program text.

Linear types enforce useful design constraints. Imagine a denotational semantics of a programming language. We would expect the store not to be duplicated or discarded—there would be something odd about a semantics that did either of these. Giving the store a linear type guarantees these properties.

Similarly, if the file system is to be represented by a single value (along the lines outlined above), then it is useful to give the file system a linear type. Among other things, this would trap some errors that novice programmers might make, such as throwing away the entire file system. (An early proposal for Haskell I/O treated the file system as a value. It was rejected, in part, because no mechanism like linearity was available.)

The system described here divides types into two families. Values of linear type have exactly one reference to them, and so require no garbage collection. Values of nonlinear type may have many pointers to them, or none, and do require garbage collection.

Pure linearity is, in fact, a stronger constraint than necessary. It is ok to have more than one reference to a value, so long as the value is being “read”; only when the value is “written” (e.g., destructively updated) is it necessary to guarantee that a single reference exists.

¹Miranda is a trademark of Research Software Limited.

For example, the “cat” program given previously is not legal as it stands. It should be written

```
main files a = let! (a)v = concat [lookup i a | i ← files] in
               update “stdout” v .
```

This allows multiple read accesses to the file system (one for each file in *files*), all of which may occur in parallel; but all of which must be completed before the “write” access (to “stdout”).

The language used in this paper is lazy. The one exception is the “**let!**” construct, which completely evaluates the term after the “**=**” before commencing evaluation of the term after the “**in**”. This sequencing is essential to guarantee that all reads of a structure are completed before it is updated. (In this example, it also has the unfortunate effect of removing the opportunity for lazy evaluation to cause reading the inputs and writing the output to act as coroutines.)

The name “single threading” was coined by Schmidt to describe the situation where the store of a denotational semantics satisfies the “no duplication” property, and he gave syntactic criteria for recognising when this occurs [Sch82, Sch85]. However, Schmidt’s criteria say nothing about the “no discarding” property, and are designed for use with a strict, rather than a lazy, language. Further, Schmidt allows only one single-threaded value (i.e., the store) while the linear type system allows any number (e.g., two different destructively updated arrays). Nonetheless, the linear type system was closely inspired by Schmidt’s work. In particular, the “**let!**” construct was added so that it would be straightforward to transform any single-threaded semantics (well, any one that doesn’t discard its store) into an equivalent program with a linear store type.

A great deal of work has gone into compile-time analysis to determine when destructive updating is safe, notably by Bloss and Hudak [Blo89, BHY89, Hud86]; older analysis techniques for determining when list cells can be reused go back to Darlington and Burstall [DB76]. Analysis techniques have the advantage that destructive updating can be inferred whether the user indicates it explicitly or not. With linear types, the user must decide which types are linear, and explicitly use **let!** where it is appropriate. Conversely, linear types have the advantage that the types make the user’s intention manifest. With analysis techniques, a small change to a program might (by fooling the analysis) result in an unintended, large change in execution efficiency whose cause is difficult to trace.

The type system used in this paper is monomorphic. It is straightforward to extend it to a polymorphic language with explicit type applications, as in the Girard-Reynolds calculus [Gir72, Gir86, Rey74, Rey83, Rey85]. However, it is not clear whether it can be extended to the more common Hindley-Milner-Damas inference system [Hin69, Mil78, DM82] used in languages such as Miranda and Haskell. This is one area for future research.

Other computer scientists have also been struck by the potential of a type system based on Girard’s linear logic.

A very elegant linear language has been developed by Lafont [Laf88], and a variant of it by Holmström [Hol88]. These systems have the amazing property that *every* value has exactly one reference to it, and so garbage collection is not required *at all*. This seems too good to be true, and perhaps it is: since there is never more than one pointer

to a value, *the results of computations cannot be shared*. In fact, sharing is allowed (the so-called “of course” types) but shared values are dealt with in two ways, neither of which is as efficient as one would like:

- The first way is to represent the shared value by a pointer to a closure. This is how Lafont’s system implements values of functional type, and how Holmström’s system implements all values. Evaluating a closure does *not* cause the closure to be overwritten. Hence, this implementation is more like call-by-name than call-by-need: any shared value will be recomputed *each time* it is used, which can be staggeringly inefficient.
- The second way is to completely copy a shared value each time a reference to it is copied. This is how Lafont’s system implements values of base type, such as lists. This is much more efficient than the first method, but it is still far less efficient than the usual conception of shared pointers.

While marvelously elegant and worthy of study, these systems seem unsuited for practical use.

The system described here is inspired in part by Lafont and Holmström’s work, but it differs in three ways:

- Lafont and Holmström use a single family of types, augmented with the “of course” type constructor. The system given here uses two completely distinct families of types, one linear and one nonlinear. This is less elegant, as it means most of the typing rules appear in two versions, one for each family. But it means one has the advantage of unique reference, for linear types, while retaining the efficiency of sharing, for nonlinear types. In particular, none of the efficiency problems above arise.
- Lafont and Holmström use syntaxes rather different than that of the lambda calculus, and a non-trivial translation is required to transform lambda calculus terms into terms in their languages. In contrast, the traditional lambda calculus is a subset of the language described here.
- Lafont and Holmström have no analogue of the “*let!*” construct, and hence do not support some useful ways of structuring programs. In particular, there is no obvious way to translate Schmidt’s single-threaded programs into linear programs in the style of Lafont and Holmström, but there is a straightforward translation into the type system in this paper.

Another type system, more loosely inspired by linear logic, has been developed by Hudak and Guzmán [HG89]. The two were developed concurrently and mostly independently, although the exact form of “*let!*” used here was partly influenced by their work. The goals of their work are more ambitious, and as a result their type system is (perhaps) more complex. Further work is needed to understand the relation between the two systems. One clear difference is that their system guarantees the “no duplication” property, but not the “no discarding” property; hence in their system deallocation of linear values is not always explicit.

The remainder of this paper is organised as follows. Section 1 describes a conventional typed language. Section 2 modifies this language for linear types, and Section 3 adds

back in nonlinear types. Section 4 introduces the “let!” construct. Section 5 presents an extended example involving arrays, showing how to implement an interpreter for a simple imperative language. Section 6 concludes.

1 Conventional types

This section presents the definition of a conventional typed lambda calculus, that is, one without linear types. The definition will be adapted to linear types in the following sections.

The only novel feature of the calculus in this section is an unusual form of data type declaration. That is, it is unusual for theorists; in practice, a quite similar form of declaration is used in languages such as Miranda and Haskell.

Assume there is a fixed set of base type declarations. Each declaration takes the form

$$K = C_1 \ T_{11} \dots T_{1k_1} \mid \dots \mid C_n \ T_{n1} \dots T_{nk_n},$$

where K is a new base type name, the C_i are new constructor names, and the T_{ij} are types (called the *immediate components* of K). For example, here are declarations for booleans and lists of Val , where Val is another base type:

$$\begin{aligned} Bool &= True \mid False, \\ List &= Nil \mid Cons \ Val \ List. \end{aligned}$$

The immediate components of $List$ are Val and $List$.

A type is a base type or a function type:

$$\begin{aligned} T ::= & K \\ & | \ (U \rightarrow V). \end{aligned}$$

Here K ranges over base types and T, U, V range over types.

A term is variable, abstraction, application, constructor term, case term, or fixpoint:

$$\begin{aligned} t ::= & x \\ & | \ (\lambda x : U. v) \\ & | \ (t u) \\ & | \ (C \ t_1 \dots t_k) \\ & | \ (\text{case } u \text{ of } C_1 \ x_{11} \dots x_{1k_1} \rightarrow v_1 \mid \dots \mid C_n \ x_{n1} \dots x_{nk_n} \rightarrow v_n) \\ & | \ (\text{fix } t). \end{aligned}$$

Here x ranges over variables and t, u, v range over terms. Following convention, parentheses may be dropped in $(T \rightarrow (U \rightarrow V))$ and $((t u) v)$.

Let A, B range over assumption lists. An assumption list associates variables with types:

$$A ::= x_1 : T_1, \dots, x_n : T_n.$$

Write $x \notin A$ to indicate that x is not one of the variables in A .

A typing is an assertion of the form $A \vdash t : T$. This asserts that if the assumptions in A are satisfied (that is, x_i has type T_i for each $x_i : T_i$ in A) then t has type T . The type system possesses *uniqueness of type*: for a given A and t there is at most one T such that $A \vdash t : T$.

Typings are derived using the rules shown in Figure 1. As usual, these rules take the form of a number of hypotheses (above the line) and a conclusion (below the line) that can be drawn if all the hypotheses are satisfied. The rules concerning base types include an additional hypothesis displaying the declaration of the type, shown in a box above the rule. The rules come in pairs: the $\rightarrow \mathcal{I}$ rule introduces a function (by a lambda expression), and the $\rightarrow \mathcal{E}$ rule eliminates a function (by applying it); the $K\mathcal{I}$ rule introduces a value of a base type (by constructing it), and the $K\mathcal{E}$ rule eliminates a value of a base type (by a case analysis). There are also rules for variables and fixpoints (VAR and FIX).

As an example, here is a function two append two lists, written out in painful detail:

```
fix (λappend : List → List → List.
    λxs : List. λys : List.
    case xs of
      Nil → ys
      Cons x' xs' → Cons x' (append xs' ys)) .
```

This term is well-typed, with type $List \rightarrow List \rightarrow List$.

2 Linear types

In the conventional type system just described, each occurrence of a variable in an assumption list, $x : T$, can be read as permission to use the variable x at type T . Furthermore, it can be used any number of times: zero, one, or many. Hence

$$x : Arr \vdash () : Unit$$

is a valid typing, even though the assumption $x : Arr$ is used zero times. Similarly,

$$x : Arr \vdash (x, x) : Arr \times Arr$$

is also valid, even though the assumption $x : Arr$ is used twice.

The key idea in a linear type system is that *each assumption must be used exactly once*. Thus both of the above typings become illegal. As a first approximation, if $A \vdash t : T$ is a valid typing in a linear type system, then each variable in A appears exactly once in t .

Linear types are written differently from conventional types. Base types are written $\mathbf{i}K$, and function types are written $U \multimap V$. (In linear logic, by default an assumption T must be used exactly once; if an assumption is to be discarded or duplicated it must be written $\mathbf{!}T$. The symbol for linear types was chosen to reflect this.)

Thus, the new grammar of types is:

$$\begin{aligned} T ::= & \mathbf{i}K \\ & | (U \multimap V) . \end{aligned}$$

The grammar of terms is changed similarly:

$$\begin{aligned} t ::= & x \\ & | (\mathbf{i}\lambda x : U. v) \\ & | (\mathbf{it} u) \\ & | (\mathbf{i}C t_1 \dots t_k) \\ & | (\mathbf{case} u \mathbf{of} \mathbf{i}C_1 x_{11} \dots x_{1k_1} \rightarrow v_1 | \dots | \mathbf{i}C_n x_{n1} \dots x_{nk_n} \rightarrow v_n) . \end{aligned}$$

$$\text{VAR} \frac{}{A, x : T \vdash x : T}$$

$$\rightarrow \mathcal{I} \frac{A, x : U \vdash v : V}{A \vdash (\lambda x : U. v) : U \rightarrow V} x \notin A$$

$$\rightarrow \mathcal{E} \frac{A \vdash t : U \rightarrow V \quad A \vdash u : U}{A \vdash (t \ u) : V}$$

$$[K = \dots | C \ T_1 \ \dots \ T_k | \dots]$$

$$A \vdash t_1 : T_1 \\ \dots \\ K \mathcal{I} \frac{A \vdash t_k : T_k}{A \vdash (C \ t_1 \ \dots \ t_k) : K}$$

$$[K = C_1 \ T_{11} \ \dots \ T_{1k_1} | \dots | C_n \ T_{n1} \ \dots \ T_{nk_n}]$$

$$A \vdash u : K \\ A, x_{11} : T_{11}, \dots, x_{1k_1} : T_{1k_1} \vdash v_1 : V \\ \dots \\ K \mathcal{E} \frac{A, x_{n1} : T_{n1}, \dots, x_{nk_n} : T_{nk_n} \vdash v_n : V}{A \vdash (\text{case } u \text{ of } C_1 \ x_{11} \ \dots \ x_{1k_1} \rightarrow v_1 | \dots | C_n \ x_{n1} \ \dots \ x_{nk_n} \rightarrow v_n) : V} x_{ij} \notin A$$

$$\text{FIX} \frac{A \vdash t : T \rightarrow T}{A \vdash (fix \ t) : T}$$

Figure 1: Conventional typing rules

$$\text{VAR} \frac{}{x : T \vdash x : T}$$

$$\text{-oI} \frac{A, x : U \vdash v : V}{A \vdash (\lambda x : U. v) : U \rightarrow V} x \notin A$$

$$\text{-oE} \frac{A \vdash t : U \rightarrow V \quad B \vdash u : U}{A, B \vdash (t u) : V}$$

$$[\mathsf{i}K = \dots \mid \mathsf{i}C \ T_1 \dots T_k \mid \dots]$$

$$\mathsf{i}K\mathcal{I} \frac{A_1 \vdash t_1 : T_1 \quad \dots \quad A_k \vdash t_k : T_k}{A_1, \dots, A_k \vdash (\mathsf{i}C \ t_1 \dots t_k) : K}$$

$$[\mathsf{i}K = \mathsf{i}C_1 \ T_{11} \dots T_{1k_1} \mid \dots \mid \mathsf{i}C_n \ T_{n1} \dots T_{nk_n}]$$

$$\mathsf{i}K\mathcal{E} \frac{A \vdash u : K \quad B, x_{11} : T_{11}, \dots, x_{1k_1} : T_{1k_1} \vdash v_1 : V \quad \dots \quad B, x_{n1} : T_{n1}, \dots, x_{nk_n} : T_{nk_n} \vdash v_n : V}{A, B \vdash (\mathbf{case} \ u \ \mathbf{of} \ \mathsf{i}C_1 \ x_{11} \dots x_{1k_1} \rightarrow v_1 \mid \dots \mid \mathsf{i}C_n \ x_{n1} \dots x_{nk_n} \rightarrow v_n) : V} x_{ij} \notin B$$

Figure 2: Rules for linear types

There are no fixpoints in the linear language. A linear value should be accessed exactly once, but $\mathit{fix} \ t$ is equivalent to $t \ (t \ (\dots))$.

Each of the typing rules must now be modified accordingly. The new rules are summarised in Figure 2.

Consider the old VAR rule:

$$\text{VAR} \frac{}{A, x : T \vdash x : T} .$$

Any assumption in the list A is unused in the typing, and hence this is no longer legal. The new rule is:

$$\text{VAR} \frac{}{x : T \vdash x : T} .$$

Here the single assumption on the left is used exactly once on the right.

Next, consider the old $\rightarrow \mathcal{E}$ rule:

$$\rightarrow \mathcal{E} \frac{A \vdash t : U \rightarrow V \quad A \vdash u : U}{A \vdash (t \ u) : V} .$$

Any variable that appears in A may appear in both t and u , and this should be prohibited. The new rule is:

$$\rightarrow \mathcal{E} \frac{A \vdash t : U \multimap V \quad B \vdash u : U}{A, B \vdash (it \ u) : V} .$$

Here both A and B range over assumption lists, and A, B stands for the conjunction of the two lists. It is easy to see that if each variable in A occurs exactly once in t , and if each variable in B occurs exactly once in u , then each variable in A, B occurs exactly once in $t \ u$.

The $iK\mathcal{I}$ rule is modified to refer to k assumption lists, A_1, \dots, A_k . On the other hand, the $iK\mathcal{E}$ rule is modified to refer to only two assumption lists, A and B . The same assumption list B is used for each of the branches of the **case**; this is sensible because exactly one branch will be executed. This, incidentally, is why the claim that each variable appears exactly once is only approximate; in a **case** term a variable may appear exactly once in each branch.

We can define some familiar combinators by writing I_X , B_{XYZ} , and C_{XYZ} as abbreviations for

$$\begin{aligned} I_X &= i\lambda x : X. x \\ B_{XYZ} &= i\lambda f : Y \multimap Z. i\lambda g : X \multimap Y. i\lambda x : X. i(f \ g \ x) \\ C_{XYZ} &= i\lambda f : X \multimap Y \multimap Z. i\lambda y : Y. i\lambda x : X. i(i(f \ x) \ y) \end{aligned}$$

yielding the well-typings

$$\begin{aligned} \vdash I_X &: X \multimap X \\ \vdash B_{XYZ} &: (Y \multimap Z) \multimap (X \multimap Y) \multimap X \multimap Z \\ \vdash C_{XYZ} &: (X \multimap Y \multimap Z) \multimap (Y \multimap X \multimap Z) . \end{aligned}$$

On the other hand, if we define

$$\begin{aligned} K_{XY} &= i\lambda x : X. i\lambda y : Y. x \\ S_{XYZ} &= i\lambda f : X \multimap Y \multimap Z. i\lambda g : X \multimap Y. i\lambda x : X. i(i(f \ x) \ (g \ x)) \end{aligned}$$

then K_{XY} and S_{XYZ} have no well-typings in the linear system; K because it discards y , and S because duplicates x . More generally, any term that translates into a combination of the I , B , and C combinators is well-typed in this system, while any term requiring K or S for its translation is not.

In the linear system, each operation that introduces (and allocates) a value is paired with exactly one operation that eliminates (and deallocates) that value. The introduction operations (abstraction and construction) create values to which a single reference exists. This reference may never be duplicated or discarded, so the elimination operations (application and case) act on values to which they hold the sole reference. Hence, after an application the storage occupied by the function may be reclaimed, and after a case analysis the storage occupied by the node analysed may be reclaimed. No reference counting or garbage collection is required.

This efficiency is achieved by restricting the language severely: each variable that is bound must be used exactly once. This is perfect for variables corresponding to, say, a file system or a large array; but it is not reasonable to impose such a restriction on a variable containing, say, an integer. Thus, the type system needs to distinguish two sorts of values: those that may not be duplicated or discarded, and those that may. This extension is the subject of the next section.

3 Nonlinear types

The linear language of the previous section is wonderfully efficient, but woefully lacking in expressiveness. To recover the power of a sensible programming language, we reintroduce the types K and $U \rightarrow V$. This yields a language with two families of types: *linear* types, $\mathbf{i}K$ and $U \multimap V$, and *nonlinear* types, K and $U \rightarrow V$. Values of linear types may not be duplicated or discarded; values of nonlinear types may.

If T is a linear type, then an assumption of the form $x : T$ is a permission, and a requirement, to use the variable x exactly once. It is a restrictive assumption. May I discard x ? $\mathbf{!No!}$ May I duplicate x ? $\mathbf{!No!}$

If T is a nonlinear type, then an assumption of the form $x : T$ is a permission to use x zero, one, or many times. It is a generous assumption. May I discard x ? Of course! May I duplicate x ? Of course!

(Similarly, in linear logic an assumption may not be discarded or duplicated unless it is of the form $!T$. The $!$ is pronounced “of course”. But be warned: the formula $U \rightarrow V = (!U) \multimap V$ that holds in linear logic is *not* appropriate to this paper; a better guide would be $U \rightarrow V = !(U \multimap V)$.)

There are now two forms of base type declaration, linear and nonlinear:

$$\begin{aligned}\mathbf{i}K &= \mathbf{i}C_1\ T_{11} \dots T_{1k_1} \mid \dots \mid \mathbf{i}C_n\ T_{n1} \dots T_{nk_n}, \\ K &= C_1\ T_{11} \dots T_{1k_1} \mid \dots \mid C_n\ T_{n1} \dots T_{nk_n}.\end{aligned}$$

In the former, the immediate components may be any types, while in the latter case they must be nonlinear. In other words, a nonlinear data structure must not contain any linear components. Thus,

$$\begin{aligned}\mathbf{i}List &= \mathbf{i}Nil \mid \mathbf{i}Cons\ \mathbf{i}Val\ \mathbf{i}List, \\ \mathbf{i}List &= \mathbf{i}Nil \mid \mathbf{i}Cons\ Val\ \mathbf{i}List, \text{ and} \\ List &= Nil \mid Cons\ Val\ List\end{aligned}$$

are all ok, but

$$List = Nil \mid Cons\ \mathbf{i}Val\ List$$

is not ok.

This restriction is easy to understand. A value of linear type must be accessed exactly once. Say that a value of nonlinear type contained a pointer to it. If the nonlinear value was duplicated, the linear value would be accessed once for each duplication; it would be virtually duplicated. If the nonlinear value was discarded, the linear value would never be accessed; it would be virtually discarded. Hence, no nonlinear value may point to a linear value.

For example, the following fragment should be well-typed if xs has type $List$:

```
case  $xs$  of ... |  $Cons\ x'\ xs' \rightarrow$ 
case  $xs$  of ... |  $Cons\ x''\ xs'' \rightarrow$ 
...  $x'\dots x''\dots$ 
```

Under the *illegal* declaration of $List$ above, both x' and x'' would have the linear type iVal , even though they are both bound to the same value. Any operation that took advantage of the linear type of x' (to reuse its storage, or to update it destructively) would create a “side effect” on x'' . So much for referential transparency!

Fortunately, the linear type system disallows this. With the legal declaration of the type $List$, both x' and x'' have the nonlinear type Val , and there is no problem. On the other hand, if xs has the linear type iList , then the fragment would be illegal because it uses xs twice.

The grammar of types is now

$$\begin{array}{lcl} T & ::= & \mathsf{iK} \\ & | & K \\ & | & (U \multimap V) \\ & | & (U \rightarrow V). \end{array}$$

Each type is deemed linear or nonlinear, depending on its topmost constructor. Hence $(T \multimap (U \rightarrow V))$ is linear, while $(T \rightarrow (U \multimap V))$ is nonlinear.

The grammar of terms is similarly extended. To all of the term formers in Section 2 are added all of the term formers in Section 1. Thus $(\mathsf{i}\lambda x : U. v)$ is a term of type $U \multimap V$, and $(\lambda x : U. v)$ is a term of type $U \rightarrow V$. The typing rules consist of all those in Figure 2 together with the additional rules in Figure 3.

Mathematically, the notion that values of a nonlinear type may be discarded or duplicated is represented by the rules:

$$\begin{array}{c} \text{KILL } \frac{A \vdash u : U}{A, x : T \vdash u : U} \text{ nonlinear } T, \\ \text{COPY } \frac{A, x : T, x : T \vdash u : U}{A, x : T \vdash u : U} \text{ nonlinear } T. \end{array}$$

The KILL rule allows a value of type T to be discarded, while the COPY rule allows a value of type T to be duplicated, so long as T is a nonlinear type. To formulate the COPY rule neatly, assumptions of the form $x : T$ are allowed to appear possibly multiple times in an assumption list when T is a nonlinear type; thus assumption lists are multisets rather than sets. Note that the VAR rule of Figure 2 applies to both linear and nonlinear types.

The nonlinear version of the function introduction rule is:

$$\rightarrow_{\mathcal{I}} \frac{A, x : U \vdash v : V}{A \vdash (\lambda x : U. v) : U \rightarrow V} x \notin A, \text{ nonlinear } A.$$

This is identical to the old rule, except for the addition of a side condition requiring that the assumption list A be nonlinear. An assumption list is nonlinear if each assumption

$$\text{KILL } \frac{A \vdash u : U}{A, x : T \vdash u : U} \text{ nonlinear } T$$

$$\text{COPY } \frac{A, x : T, x : T \vdash u : U}{A, x : T \vdash u : U} \text{ nonlinear } T$$

$$\rightarrow \mathcal{I} \frac{A, x : U \vdash v : V}{A \vdash (\lambda x : U. v) : U \rightarrow V} x \notin A, \text{ nonlinear } A$$

$$\rightarrow \mathcal{E} \frac{A \vdash t : U \rightarrow V \quad B \vdash u : U}{A, B \vdash (t \ u) : V}$$

$$[K = \dots \mid C \ T_1 \ \dots \ T_k \mid \dots, \quad \text{nonlinear } T_i]$$

$$\begin{array}{c} A_1 \vdash t_1 : T_1 \\ \cdots \\ K\mathcal{I} \frac{A_k \vdash t_k : T_k}{A_1, \dots, A_k \vdash (C \ t_1 \ \dots \ t_k) : K} \end{array}$$

$$[K = C_1 \ T_{11} \ \dots \ T_{1k_1} \mid \dots \mid C_n \ T_{n1} \ \dots \ T_{nk_n}, \quad \text{nonlinear } T_{ij}]$$

$$K\mathcal{E} \frac{\begin{array}{c} A \vdash u : K \\ B, x_{11} : T_{11}, \dots, x_{1k_1} : T_{1k_1} \vdash v_1 : V \\ \cdots \\ B, x_{n1} : T_{n1}, \dots, x_{nk_n} : T_{nk_n} \vdash v_n : V \end{array}}{A, B \vdash (\text{case } u \text{ of } C_1 x_{11} \ \dots \ x_{1k_1} \rightarrow v_1 \mid \dots \mid C_n x_{n1} \ \dots \ x_{nk_n} \rightarrow v_n) : V} x_{ij} \notin B$$

$$\text{FIX } \frac{A \vdash t : T \rightarrow T}{A \vdash (\text{fix } t) : T}$$

Figure 3: Rules for nonlinear types

$x_i : T_i$ in it has a nonlinear T_i . No condition is placed on U or V by this rule; they may be either linear or nonlinear.

This restriction follows from the principle, established above, that a nonlinear value must not contain pointers to linear values. A function value is a closure that contains a pointer to an environment binding each variable in A . Hence a nonlinear function can only be introduced in an environment A that contains only nonlinear types.

For example, if $\mathbf{i}X$ is a linear type and Y is nonlinear, then

$$(\lambda x : \mathbf{i}X. \lambda y : Y. x) : \mathbf{i}X \rightarrow Y \rightarrow \mathbf{i}X$$

is not a well-typing. (It is an easy exercise to show that if it were well-typed then any linear value could be duplicated.) On the other hand, both of

$$\begin{aligned} &\vdash (\lambda x : \mathbf{i}X. \mathbf{i}\lambda y : Y. x) : \mathbf{i}X \rightarrow Y \multimap \mathbf{i}X, \\ &\vdash (\lambda y : Y. \lambda x : \mathbf{i}X. x) : Y \rightarrow \mathbf{i}X \rightarrow \mathbf{i}X \end{aligned}$$

are ok, the first because $\multimap \mathcal{I}$ places no constraint on the assumption list, and the second because $\rightarrow \mathcal{I}$ places no constraint on the argument or result type.

Returning to a previous example, here is another version of the append function:

```
fix (λappend : |List →o |List →o |List.
    λxs : |List. λys : |List.
      case xs of
        |Nil → ys
        |Cons x' xs' → |Cons x' (i(iappend xs') ys) ) .
```

This is well-typed under either declaration for $|List$ given above. Since both arguments are linear, if the first argument is a $|Cons$ cell then this cell may be deallocated immediately. It requires only a modestly good compiler to notice that the best thing to do with this cell is not to return it to free storage, but to reuse it in building the result. An only moderately better compiler would notice that the head of this cell may be left unchanged. Only the tail of the cell needs to be filled in with the result of the recursive call to $append$. In fact, that recursive call will almost always return the same value that is there already (the exception being when it is $|Nil$), and a little loop unrolling could result in a very efficient version of $append$ indeed—but it might require a less modestly good compiler writer to notice that.

4 Read-only access

In order for destructive updating of a value to be safe, it is essential that there be only one reference to the value when the update occurs. In the linear type system, this is enforced by guaranteeing that there is always exactly one reference to the value. This restriction is stronger than necessary. It is perfectly safe to have more than one reference to a value temporarily, as long as only one reference exists when the update is performed.

The situation here is similar to the well-known “readers and writers” problem, where access to a resource is to be controlled. Many processes may simultaneously have read access, but a process may have write access only if no other process has access (either read or write) to the resource.

The two sorts of access are modelled using two types. A linear type corresponds to write access, and a nonlinear type corresponds to read access: permission to write must be unique, but permission to read may be freely duplicated.

A new form of term is introduced for granting read-only access:

$$\text{let! } (x) y = u \text{ in } v .$$

This is evaluated similarly to a conventional **let** term: first u is evaluated, then y is bound to the result, then v is evaluated in the extended environment. But there are three differences from a garden-variety **let**.

The first is that within u the variable x has nonlinear type, while within v the variable x has linear type. That is, during evaluation of u , read-only access is allowed to the value in x .

The second is that evaluation of u must be carried out completely before evaluation of v commences—this is sometimes called *hyperstrict* evaluation. For instance, if u returns a list, then all elements of this list must be evaluated. This is required in order to guarantee that all references to x within u are freed before evaluation of v commences.

The third is that there are some constraints on the type of x and u . It must not be possible for u (or any component of u) to be equal to x (or any component of x)—otherwise, read access to x (or a component of x) could be passed outside the scope of u . This condition is made precise below.

Define the *components* of a type to be itself and, if it is a base type, all components of its immediate components. We will restrict our attention to the case where all components of the type of x are base types.

Given a type T , the corresponding nonlinear type $!T$ is derived from it as follows. If iK is a linear base type defined by

$$iK = iC_1 \ T_{11} \dots T_{1k_1} \mid \dots \mid iC_n \ T_{n1} \dots T_{nk_n},$$

then $!iK$ is the nonlinear base type K defined by

$$K = C_1 \ !T_{11} \dots !T_{1k_1} \mid \dots \mid C_n \ !T_{n1} \dots !T_{nk_n},$$

where the components of K are recursively converted. If K is a nonlinear base type, then $!K$ is identical to K .

Let T be the type of x , and U be the type of u . We must ensure that u cannot “smuggle out” any component of x that is linear. This is guaranteed if no linear component of T has a corresponding nonlinear component in U , and if no component of U is a function type. (Functions are disallowed since a function term may have x or a component of x as a free variable.) If this holds, say that U is *safe* for T .

Finally, the required typing rule is:

$$\text{LET! } \frac{\begin{array}{c} A, x : !T \vdash u : U \\ B, x : T, y : U \vdash v : V \end{array}}{A, B, x : T \vdash (\text{let! } (x) y = u \text{ in } v) : V} \quad U \text{ safe for } T .$$

An example of the use of **let!** appears in the next section.

It is easy to allow read-only access to several variables at once; simply take

$$\text{let! } (x_1, x_2, \dots, x_m) y = u \text{ in } v$$

as an abbreviation for

$$\text{let! } (x_1) y = (\text{let! } (x_2, \dots, x_m) y' = u \text{ in } y') \text{ in } v .$$

5 Arrays

5.1 Conventional arrays

Before considering how to add arrays to a linear type system, let's review the use of arrays in the conventional type system.

An array associates indices with values. We will fix the index and value types, and write Arr for the type of arrays with indices of type Ix and values of type Val . There are three operations of interest on arrays:

$$\begin{aligned} \text{alloc} &: \text{Arr}, \\ \text{lookup} &: \text{Ix} \rightarrow \text{Arr} \rightarrow \text{Val}, \\ \text{update} &: \text{Ix} \rightarrow \text{Val} \rightarrow \text{Arr} \rightarrow \text{Arr}. \end{aligned}$$

Here alloc allocates a new array (with all entries set to some fixed initial value); $\text{lookup } i \ a$ returns the value at index i in array a ; and $\text{update } i \ v \ a$ returns an array identical to a except that index i is associated with value v . (In practice, the array type may be parameterised on the index and value types, and the new array function may take additional arguments to determine index bounds and initial values; but the simpler version suffices to demonstrate the central ideas.)

This section will use the small interpreter shown in Figure 4 as a running example. This is written in an equational notation familiar from languages such as Miranda and Haskell; it is easy to translate the equations into lambda and case terms.

The interpreter can be read as a denotational semantics for a simple imperative language. Variable names are identified with type Ix , the values stored in variables are identified with type Val , and stores (which map variable names to values) are identified with type Arr .

Three data types correspond to the abstract syntax of expressions, commands, and programs.

- An expression is a variable, a constant, or the sum of two expressions. The semantic function corresponding to an expression takes an array into a value.
- A command is an assignment, a sequence of two commands, or a conditional. The semantic function corresponding to a command takes an array into an array.
- A program consists of a command followed by an expression. The semantics corresponding to a program is a value.

The interpreter satisfies Schmidt's single-threading criteria. Thus, it is safe to implement the update operation in a way that re-uses the store allocated for the array. (This is to be expected, since the array represents the store of an imperative language.) However, how is the implementation to determine when it is safe to implement update in this way? Succeeding sections will show how linear types can be used for this purpose.

5.2 Linear arrays

Now that the framework of the linear type system is in place, adding primitives for arrays is relatively straightforward. This section will show how to add arrays when

$Expr$	$= Var \ Ix \mid Const \ Val \mid Plus \ Expr \ Expr$
Com	$= Asgn \ Ix \ Expr \mid Seq \ Com \ Com \mid If \ Expr \ Com \ Com$
$Prog$	$= Do \ Com \ Expr$
$expr$	$: Expr \rightarrow Arr \rightarrow Val$
$expr \ (Var \ i) \ a$	$= lookup \ i \ a$
$expr \ (Const \ v) \ a$	$= v$
$expr \ (Plus \ e_0 \ e_1) \ a$	$= expr \ e_0 \ a + expr \ e_1 \ a$
com	$: Com \rightarrow Arr \rightarrow Arr$
$com \ (Asgn \ i \ e) \ a$	$= update \ i \ (expr \ e \ a) \ a$
$com \ (Seq \ c_0 \ c_1) \ a$	$= com \ c_1 \ (com \ c_0 \ a)$
$com \ (If \ e \ c_0 \ c_1) \ a$	$= \text{if } expr \ e \ a = 0 \text{ then } com \ c_0 \ a \text{ else } com \ c_1 \ a$
$prog$	$: Prog \rightarrow Val$
$prog \ (Do \ c \ e)$	$= expr \ e \ (com \ c \ alloc)$

Figure 4: A simple interpreter

“read only” access is not used. This leads to a small problem, which will be resolved by the use of “read only” access in the next section.

As before, an array type maps indices into values. The array type is linear, and so is written \mathbf{iArr} . For the simplest version of arrays, the index and value types are nonlinear, and so are written Ix and Val .

We will require a data type that pairs a Val with a \mathbf{iArr} . This may be declared by

$$\mathbf{iValArr} = \mathbf{iMkPair} \ Val \ \mathbf{iArr},$$

but for readability we will write $Val \otimes \mathbf{iArr}$ instead of $\mathbf{iValArr}$, and (v, a) instead of $\mathbf{iMkPair} \ v \ a$, and $\mathbf{let} \ (v, a) = t \ \mathbf{in} \ u$ instead of $\mathbf{case} \ t \ \mathbf{of} \ \mathbf{iMkPair} \ v \ a \rightarrow u$.

The four operations on linear arrays are:

$$\begin{aligned} alloc &: \mathbf{iArr}, \\ lookup &: Ix \rightarrow \mathbf{iArr} \rightarrow Val \otimes \mathbf{iArr}, \\ update &: Ix \rightarrow Val \rightarrow \mathbf{iArr} \rightarrow \mathbf{iArr}, \\ dealloc &: Val \otimes \mathbf{iArr} \rightarrow Val. \end{aligned}$$

The $alloc$ and $update$ operations are as before. The new $lookup$ operation, being passed the sole reference to an array, must return a reference to the same array when it completes. Otherwise, since arrays cannot be duplicated, an array could only be indexed once, and then would be lost forever! The linear type discipline requires that values of a linear type be explicitly deallocated, and this is the purpose of the $dalloc$ operation.

The meaning of these operations can be explained in terms of the old operations as

$Expr$	$= Var\ Ix \mid Const\ Val \mid Plus\ Expr\ Expr$
Com	$= Asgn\ Ix\ Expr \mid Seq\ Com\ Com \mid If\ Expr\ Com\ Com$
$Prog$	$= Do\ Com\ Expr$
$expr$	$: Expr \rightarrow \text{iArr} \rightarrow Val \otimes \text{iArr}$
$expr (Var i) a$	$= lookup i a$
$expr (Const v) a$	$= (v, a)$
$expr (Plus e_0 e_1) a$	$= \text{let } (v_0, a_0) = expr e_0 a \text{ in}$ $\quad \text{let } (v_1, a_1) = expr e_1 a_0 \text{ in}$ $\quad (v_0 + v_1, a_1)$
com	$: Com \rightarrow \text{iArr} \rightarrow \text{iArr}$
$com (Asgn i e) a$	$= \text{let } (v, a') = expr e a \text{ in } update i v a'$
$com (Seq c_0 c_1) a$	$= com c_0 (com c_1 a)$
$com (If e c_0 c_1) a$	$= \text{let } (v, a') = expr e a \text{ in}$ $\quad \text{if } v = 0 \text{ then } com c_0 a' \text{ else } com c_1 a'$
$prog$	$: Prog \rightarrow Val$
$prog (Do c e)$	$= dealloc (expr e (com c alloc))$

Figure 5: A simple interpreter, version 2

follows:

$$\begin{aligned} alloc &= alloc_{old}, \\ lookup i a &= (lookup_{old} a i, a), \\ update i v a &= update_{old} i v a, \\ dealloc x &= \text{let } (v, a) = x \text{ in } v. \end{aligned}$$

However, we could not write this program to define the new operations; they must be defined as primitives. (The definition of *lookup* is illegal because although *a* is linear it appears twice on the right-hand side; and the definition of *dealloc* is illegal because although *a* is linear it appears no times on the right-hand side.)

A new version of the interpreter using the linear array operations is shown in Figure 5. Unfortunately, this second version is a little more elaborate than the first, because each application of *lookup* (and, hence, also each application of *expr*) requires additional plumbing to pass around the unchanged array. This is particularly unfortunate in the *Plus* case of the definition of *expr*. It should be possible to evaluate the two summands, e_0 and e_1 , in parallel, but it is necessary here to specify some order, in this case e_0 before e_1 .

The linear type system is “too linear” in that it forces a linear order to be specified for operations (like *Plus* in *expr*) that should not require it. The next section shows how “read only” access can solve this problem.

$Expr$	$= Var\ Ix \mid Const\ Val \mid Plus\ Expr\ Expr$
Com	$= Asgn\ Ix\ Expr \mid Seq\ Com\ Com \mid If\ Expr\ Com\ Com$
$Prog$	$= Do\ Com\ Expr$
$expr$	$: Expr \rightarrow Arr \rightarrow Val$
$expr\ (Var\ i)\ a$	$= lookup\ i\ a$
$expr\ (Const\ v)\ a$	$= v$
$expr\ (Plus\ e_0\ e_1)\ a$	$= expr\ e_0\ a + expr\ e_1\ a$
com	$: Com \rightarrow \text{iArr} \rightarrow \text{iArr}$
$com\ (Asgn\ i\ e)\ a$	$= \text{let! } (a) v = expr\ e\ a \text{ in update } i\ v\ a$
$com\ (Seq\ c_0\ c_1)\ a$	$= com\ c_1\ (com\ c_0\ a)$
$com\ (If\ e\ c_0\ c_1)\ a$	$= \text{let! } (a) v = expr\ e\ a \text{ in}$ $\quad \text{if } v = 0 \text{ then } com\ c_0\ a \text{ else } com\ c_1\ a$
$prog$	$: Prog \rightarrow Val$
$prog\ (Do\ c\ e)$	$= dealloc\ (expr'\ e\ (com\ c\ alloc))$
$expr'$	$: Expr \rightarrow \text{iArr} \rightarrow Val \otimes \text{iArr}$
$expr'\ e\ a$	$= \text{let! } (a) v = expr\ e\ a \text{ in } (v, a)$

Figure 6: A simple interpreter, version 3

5.3 “Read only” arrays

If “read only” access is allowed, then the array operations are as follows:

$$\begin{aligned} alloc &: \text{iArr}, \\ lookup &: Ix \rightarrow Arr \rightarrow Val, \\ update &: Ix \rightarrow Val \rightarrow \text{iArr} \rightarrow \text{iArr}, \\ dealloc &: Val \otimes \text{iArr} \rightarrow Val. \end{aligned}$$

The type iArr corresponds to write access, and the type Arr corresponds to read access. The $alloc$, $update$, and $dalloc$ operations are just as in the previous section; while the $lookup$ operation is just as it was originally in the conventional type system.

A third version of the interpreter using “read only” access is shown in Figure 6. Here the com semantic function has the same structure it had in version 2, while the $expr$ semantic function has the same structure it had in version 1. In particular, the spurious imposition of an evaluation order on summands, present in version 2, has gone away. Note that the type of $expr$ refers to Arr , not iArr , making it explicit that $expr$ never changes the array that it is passed.

One unusual feature of this simple semantics is that it does, in effect, discard the store (the command is executed; the expression is evaluated in the resulting store; and then the store is discarded and only the value of the expression is kept). Note that the “no discard” rule does not mean that the store cannot be discarded, simply that the

store must be discarded *explicitly*. As a result, the definition of *prog* in the final program (Figure 6) is more longwinded than the equivalent definition in the original (Figure 4). Depending on your point of view, this difference may be regarded as a drawback or as a benefit.

5.4 Arrays of arrays

Finally, we briefly consider how to handle arrays of arrays. Let iArr and Arr be as in the previous section: arrays taking indexes of type Ix into values of type Val . Let iArr2 and Arr2 be arrays taking indexes of type Ix into values of type iArr . The new feature here is that the values of Arr2 are linear rather than nonlinear.

Let the operations on the type Arr be as in the previous section. In addition, the operations on Arr2 are:

$$\begin{aligned} \text{alloc2} &: \text{iArr2} \\ \text{lookup2} &: \text{Ix} \rightarrow \text{Arr2} \rightarrow \text{Arr} \\ \text{update2} &: \text{Ix} \rightarrow (\text{iArr} \rightarrow \text{iArr}) \rightarrow \text{iArr2} \rightarrow \text{iArr2} \\ \text{dealloc2} &: \text{Val} \otimes \text{iArr2} \rightarrow \text{Val} \end{aligned}$$

To compute the value of $a2[k][l]$ and store this in location $a2[i][j]$ one writes:

```
let! (a2) v = lookup l (lookup2 k a2) in
  update2 i (update j v) a2 .
```

6 Conclusions

Much further work remains to be done. Among the important questions are the following:

- How do linear types fit in with the Hindley-Milner-Damas approach to polymorphism and type inference?
- How can linear types best support i/o and interactive functional programs?
- Girard’s Linear Logic contains nothing like the “let!” construct. Is there a nice theoretical justification for this construct?
- Deforestation [Wad88] has a linearity constraint, and the “blazed” types described there might be subsumed by “of course” types. Can linear types aid program transformation?

Finally, more practical experience is required before we can evaluate the likelihood that linear types *will* change the world.

Acknowledgements. For discussions relating to this work, and comments on it, I am grateful to Steve Blott, Kei Davis, Sören Holmström, Paul Hudak, Simon Jones, Yves Lafont, Simon Peyton Jones, David Schmidt, the Glasgow FP Group, and the members of IFIP WG 2.8.

References

- [Blo89] A. Bloss, Path analysis: using order-of-evaluation information to optimize lazy functional languages. Ph.D. thesis, Yale University, Department of Computer Science, 1989.
- [BHY89] A. Bloss, P. Hudak, and J. Young, An optimising compiler for a modern functional language. *Computer Journal*, **32**(2):152–161, April 1989.
- [DB76] J. Darlington and R. M. Burstall, A system which automatically improves programs. *Acta Informatica*, **6**:41–60, 1976.
- [DM82] L. Damas and R. Milner, Principal type schemes for functional programs. In *Proceedings of the 9'th Annual Symposium on Principles of Programming Languages*, Albuquerque, N.M., January 1982.
- [Gir72] J.-Y. Girard, *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. Ph.D. thesis, Université Paris VII, 1972.
- [Gir86] J.-Y. Girard, The system F of variable types, fifteen years later. *Theoretical Computer Science*, **45**:159–192, 1986.
- [Gir87] J.-Y. Girard, Linear logic. *Theoretical Computer Science*, **50**:1–102, 1987.
- [GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*. Cambridge University Press, 1989.
- [Hin69] R. Hindley, The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.*, **146**:29–60, December 1969.
- [Hol88] S. Holmström, A linear functional language. Draft paper, Chalmers University of Technology, 1988.
- [Hud86] P. Hudak, A semantic model of reference counting and its abstraction. In *Proceedings ACM Conference on Lisp and Functional Programming*, August 1986.
- [HG89] P. Hudak and J. Guzmán, Taming side effects with a single-threaded type system. Draft paper, Yale University, December 1989.
- [HW88] P. Hudak and P. Wadler, editors, *Report on the Functional Programming Language Haskell*. Technical Report YALEU/DCS/RR656, Yale University, Department of Computer Science, December 1988; also Technical Report, Glasgow University, Department of Computer Science, December 1988.
- [Laf88] Y. Lafont, The linear abstract machine. *Theoretical Computer Science*, **59**:157–180, 1988.
- [Mil78] R. Milner, A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, **17**:348–375, 1978.

- [Rey74] J. C. Reynolds, Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, LNCS 19, Springer-Verlag.
- [Rey83] J. C. Reynolds, Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, 513–523, North-Holland, Amsterdam.
- [Rey85] J. C. Reynolds, Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.
- [Sch82] D. A. Schmidt, Denotational semantics as a programming language. Internal report CSR-100, Computer Science Department, University of Edinburgh, 1982.
- [Sch85] D. A. Schmidt, Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems*, 7:299–310, 1985. (Also Internal Report CSR-143, Computer Science Department, University of Edinburgh, September 1983.)
- [Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.
- [Wad88] P. Wadler, Deforestation: transforming programs to eliminate trees. In *European Symposium on Programming*, LNCS 300, Springer-Verlag, 1988.

Separation Logic and Abstraction

Matthew Parkinson
University of Cambridge
Computer Laboratory
Cambridge CB3 0FD, UK
mjp41@cl.cam.ac.uk

Gavin Bierman
Microsoft Research
7 J J Thomson Ave
Cambridge CB3 0FB, UK
gmb@microsoft.com

ABSTRACT

In this paper we address the problem of writing specifications for programs that use various forms of modularity, including procedures and Java-like classes. We build on the formalism of separation logic and introduce the new notion of an *abstract predicate* and, more generally, abstract predicate families. This provides a flexible mechanism for reasoning about the different forms of abstraction found in modern programming languages, such as abstract datatypes and objects. As well as demonstrating the soundness of our proof system, we illustrate its utility with a series of examples.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Program Verification—*class invariants*; D.3.3 [Programming Languages]: Language Constructs and Features—*Modules, packages*; D.3.3 [Programming Languages]: Language Constructs and Features—*Classes and inheritance*

General Terms

Languages, Theory, Verification

Keywords

Separation Logic, Modularity, Resources, Abstract data types, Classes

1. INTRODUCTION

In order to assist programmers in building complex software systems, programming languages offer various forms of abstraction. In this paper we focus on those that provide some form of modularity. These range from simple procedures with local state, through abstract datatypes (ADTs), to the complexities of Java-like class hierarchies with method overriding and runtime resolution of method invocation.

Our aim is to provide intuitive ways for programmers to specify the behaviour of their modular code. Previous solutions to handling modularity are either too weak, in that certain natural specifications can not be expressed; or too strong, in that the programmer is forced to accept an unreasonable proof or annotation burden.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'05, January 12–14, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-830-X/05/0001 ...\$5.00.

We choose to build upon the recent formalism of separation logic, which facilitates local reasoning about code [23]. This local reasoning approach has proved successful when considering many real world algorithms, including the Schorr-Waite graph marking algorithm [31] and a copying garbage collector [4].

Until recently, the work on separation logic has focused exclusively on low-level C-like languages with no support for abstraction. O'Hearn, Reynolds and Yang [22] have recently added static modularity to separation logic. They hide the internal resources of a module from its clients using the so called hypothetical frame rule. This partitioning of resources between the client and the module allows them to model “ownership transfer”, where state can safely be transferred between the module and the client without fear of dereferencing dangling pointers. This allows them to reason about examples such as a simple memory manager, which allocates fixed size blocks of memory, and a queue.

Though this is a significant advance, their work is severely limited as it only models static modularity. Their modules are based on Parnas' work on information hiding [25], which deals with single instances of the hidden data structure. Hence, it can not be used for many common forms of abstraction, including ADTs and classes, where we require multiple instances of the hidden resource. For example, one would expect, given a list module, to use multiple lists in an application; and one frequently creates new objects in object-oriented applications.

Let us review the problem: take a piece of code that we wish to consider “abstract” (this could be because the code is a procedure, a module or a method). A specification is then a contract between the code and its callers. It includes a precondition that expresses what a caller must establish before the code may be executed. The implementation of the module can assume the precondition on entry. A specification also contains a postcondition that records what must hold upon exit of the module. Consequently the caller can assume the postcondition upon return from the module. When reasoning about the module and the calls, only the contract given by the specification is used: that is, we expect the appropriate form of information hiding.

Various researchers have proposed enriching the logic to view the data abstractly (as in data groups [14]), or the methods/procedures abstractly (as in method groups [30, 13]). In contrast, we propose to add the abstraction to the logical framework itself, by introducing the notion of an *abstract predicate*. An abstract predicate has a name, a definition, and a scope. Within the scope one can freely swap between using the abstract predicate's name and its definition, but outside its scope it must be handled atomically, i.e. by its name. Thus the scope defines the abstraction boundary for the abstract predicate.

In various work on separation logic (e.g. [29]) it is common

to use inductively defined predicates to represent data types. In essence we allow predicates to additionally encapsulate state and not just represent it. This gives us two key advantages: (1) the impact of changing a predicate is easy to define; and (2) by encapsulating state we are able to reason about ownership transfer.

Whilst the notion of abstract predicates is sufficient to reason about modules and simple ADTs, we should like to reason about object-oriented forms of abstractions; more precisely Java-like classes and inheritance. This adds an additional complication: not only do we have to reason about encapsulation but also inheritance. Rather pleasingly this again can be provided by reflecting the abstraction in the logical framework itself. Here the key observation is that an object can exist at multiple types through the class hierarchy. We reflect this in the logic by generalising abstract predicates to *families* of abstract predicates that are indexed by class.

The rest of the paper is structured as follows. In §2 we give a brief overview of separation logic, detailing the features that we use in this paper. In §3 we present more formally the notion of an *abstract predicate*, giving proof rules and outlining a soundness proof. We also give a number of worked examples. In §4 we extend these reasoning principles to a core subset of Java. Again we outline a soundness proof and give examples. We conclude in §5 with a comparison to related work and propose some future work.

2. SEPARATION LOGIC PRIMER

In this section we give some brief details of the fragment of separation logic that we shall use. Space prevents us giving a complete description or explanation of the significant advantages of using separation logic. The interested reader can read further details and references in a survey paper by Reynolds [29].

Separation logic is an extension to Hoare logic that permits reasoning about shared mutable state. It extends Hoare logic by adding spatial connectives to the assertion language, which allow assertions to define separation between parts of the heap. This separation provides the key feature of separation logic—*local reasoning*—specifications need only mention the state they access [23].

We use the standard model of state from separation logic. A heap, H , is a partial function from locations to values (for simplicity we take Values to be the integers and Locations to be the positive integers).

$$\mathcal{H} \stackrel{\text{def}}{=} \text{Locations} \rightarrow_{fin} \text{Values}$$

This has a partial commutative monoid for disjoint function composition:

$$H_1 * H_2 \stackrel{\text{def}}{=} \lambda l. \begin{cases} H_1(l) & l \in \text{dom}(H_1) \\ H_2(l) & l \in \text{dom}(H_2) \end{cases}$$

which is defined iff $\text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset$. A stack, S , is a function from (program) variables to values. Unlike other presentations [22], we do not interpret auxiliary variables¹ using the stack but we define an auxiliary stack, I , that is a function from auxiliary variable names to values.²

$$\begin{aligned} \mathcal{S} &\stackrel{\text{def}}{=} \text{Vars} \rightarrow \text{Values} \\ \mathcal{I} &\stackrel{\text{def}}{=} \text{AuxVars} \rightarrow \text{Values} \end{aligned}$$

We define a state as a triple consisting of a stack, a heap and an auxiliary stack. A predicate is just a set of states, and formulae are given by the following grammar where B and E range over boolean- and integer-valued expressions respectively (these are defined formally in the §3.1).

¹Sometimes called ghost or logical variables.

²We add this as we make heavy use of local variables, and do not have global variables.

$$\begin{aligned} P, Q &::= B \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \\ &\quad \mid \text{empty} \mid P * Q \mid P \dashv Q \mid E \mapsto E' \end{aligned}$$

The usual classical connectives ($\neg, \vee, \wedge, \Rightarrow$) are interpreted using the boolean algebra structure induced on the powerset of states. In addition to the boolean connectives we have the new spatial connectives $*$ and \dashv , along with the predicates *empty* and \mapsto . Taking these in reverse order: the predicate $E \mapsto E'$ consists of all the triples (S, H, I) where the heap, H , consists of the single mapping from the location given by the meaning of E to the value given by the meaning of E' .

$$S, H, I \models E \mapsto E' \stackrel{\text{def}}{=} \text{dom}(H) = \{\llbracket E \rrbracket_{S,I}\} \wedge H(\llbracket E \rrbracket_{S,I}) = \llbracket E' \rrbracket_{S,I}$$

We use the shorthand $E \mapsto E_1, E_2$ to mean $E \mapsto E_1 * E + 1 \mapsto E_2$.

The spatial conjunction $P * Q$ means the heap can be split into two disjoint parts in which P and Q hold respectively.

$$S, H, I \models P * Q \stackrel{\text{def}}{=}$$

$$\exists H_1, H_2. H_1 * H_2 = H \wedge S, H_1, I \models P \wedge S, H_2, I \models Q$$

Heaps of more than one element are specified by using the $*$ to join smaller heaps. The $*$ has a unit *empty* that consists of all states (S, H, I) where H is the empty heap. The adjunct to $*$, written \dashv , is not used in this paper so we shall suppress its (routine) definition.

The essence of “local reasoning” is that to understand how a piece of code works it should only be necessary to reason about the memory the code actually accesses (its so-called “footprint”). Ordinarily aliasing precludes such a principle but the separation enforced by the $*$ connective allows this intuition to be captured formally by the following rule.

FRAME RULE

$$\frac{\vdash \{P\}C\{Q\}}{\vdash \{P * R\}C\{Q * R\}}$$

where C does not modify the free variables of R , i.e. $\text{modifies}(C) \cap FV(R) = \emptyset$.

The side-condition is required because $*$ only describes the separation of heap locations and not variables; see [5] for more details.

Note: $\text{modifies}(C)$ denotes the set of stack variables assigned by a given command, C , e.g. $\text{modifies}(x=3) = \{x\}$. However assignment through a stack variable to the heap is not counted: $\text{modifies}([x]=3) = \emptyset$. See [31] for full definition.

By using this rule, a local specification concerning only the variables and parts of the heap that are used by C can be arbitrarily extended as long as the extension’s free variables are not modified by C . Thus, from a local specification we can infer a global specification that is appropriate to the larger footprint of an enclosing program.

3. A LANGUAGE WITH MODULES

In this section we consider reasoning about a simple imperative language with first-order functions/procedures, which is essentially the same as that considered by Reynolds [29]. To simplify the presentation we delay using Java to §4. We introduce our novel concept of an abstract predicate, and state some rules for its use. (These rules are proved sound in §3.4.) We demonstrate the power and elegance of abstract predicates in reasoning about modular code by considering two detailed examples: a connection pool and a memory manager.

3.1 Syntax

The syntax for the programming language considered in this section is given by the grammar in Figure 1. We use x to range over

```

C   :=  let  $k_1 \bar{x}_1 = C_1, \dots, k_n \bar{x}_n = C_n$  in C
      | return E |  $x = k(\bar{E})$  | newvar  $x; C$  |  $x = E$ 
      |  $x = [E]$  |  $[E] = E$  |  $x = \text{cons}(\bar{E})$  | dispose(E)
      | if B then C else C | while B C | C; C
E   :=   $x | E + E | E - E | E * E | n | \text{null}$ 
B   :=   $E == E | E \leq E | \text{true} | \text{false}$ 

```

Figure 1: Module language syntax

program variable names, and k ranges over function names. We have a distinguished program variable ret that is not modifiable except with the return command. We restrict our consideration to well-formed programs: e.g. a well-formed program only has returns as the last command of a function; and defines a function name at most once in a let . In the examples of §3.3 we will use syntactic sugar for procedures: procedure definitions are functions that return null , and procedure calls are function calls assigned to an unused variable.

The command $\text{newvar } x; C$ defines a new local variable for the command C , we use a shorthand $\text{newvar } x, \dots, y; C$ for introducing multiple variables; $x = \text{cons}(\bar{E})$ allocates $|\bar{E}|$ consecutive heap locations with the values of \bar{E} . The location E is disposed using dispose ; updated to E' with $[E] = E'$; and stored in x with $x = [E]$.

3.2 Proof rules

For the assertion language we take the language given in §2 and extend it with predicates. Naturally we restrict our consideration to well-formed formulae, and again we elide the obvious definition. We write α to range over predicate names and use a function $\text{arity}()$ from predicate names to their arity.

A judgement in our assertion language is written as follows:

$$\Lambda; \Gamma \vdash \{P\}C\{Q\}$$

This is read: the command, C , satisfies the specification $\{P\} \rightarrow \{Q\}$, given the function hypotheses, Γ , and predicate definitions, Λ . The hypotheses and definitions are given by the following grammar:

$$\begin{aligned} \Gamma &:= \epsilon \mid \{P\}k(\bar{x})\{Q\}, \Gamma \\ \Lambda &:= \epsilon \mid \alpha(\bar{x}) \stackrel{\text{def}}{=} P, \Lambda \end{aligned}$$

However, when it simplifies the presentation, we will treat Λ as a partial function from predicate names to formulae, and Γ as a partial function from function names to specifications. We define $\Lambda(\alpha)[\bar{E}]$ as $P[\bar{E}/\bar{x}]$ where Λ contains $\alpha(\bar{x}) \stackrel{\text{def}}{=} P$.

For the hypotheses, Γ , to be well-formed each function, k , can appear at most once; and the specification's free program variables are contained in its arguments and ret . For the predicate definitions, Λ , to be well-formed we require that each predicate, α , is contained at most once; the free variables of the body, P , are contained in the arguments, \bar{x} ; and P is a positive formula.³ We will only consider well-formed Γ and Λ .

Intuitively, the predicates are used like abstract data types. Abstract data types have a name, a scope and a concrete representation. Within this scope the name and the representation can be freely exchanged, but outside only the name can be used. Similarly abstract predicates have a name and a formula. The formula is scoped: inside the scope the name and the body can be exchanged,

³A positive formula is one where predicate names appear only under an even number of negations. This ensures that a fixed point can be found; this is explained in further detail in §3.4.2

and outside the predicate must be treated atomically. Hence our first rule:

ABSTRACT FUNCTION DEFINITION

$$\frac{\begin{array}{c} \Lambda, \Lambda'; \Gamma \vdash \{P_1\}C_1\{Q_1\} \\ \vdots \\ \Lambda; \Gamma, \{P_1\}k_1(\bar{x}_1)\{Q_1\}, \dots, \{P_n\}k_n(\bar{x}_n)\{Q_n\} \vdash \{P\}C\{Q\} \end{array}}{\Lambda; \Gamma \vdash \{P\}\text{let } k_1 \bar{x}_1 = C_1, \dots, k_n \bar{x}_n = C_n \text{ in } C\{Q\}}$$

- P, Q, Γ and Λ do not contain the predicate names in $\text{dom}(\Lambda')$;
- $\text{dom}(\Lambda)$ and $\text{dom}(\Lambda')$ are disjoint; and
- the functions only modify local variables: $\text{modifies}(C_i) = \emptyset (1 \leq i \leq n)$.

This rule allows a module writer to use the definition of an abstract predicate, yet the client can only use the abstract predicate name. The functions k_1, \dots, k_n are within the scope of the predicates defined in Λ' hence verifying the function bodies $C_1 \dots C_n$ can use the predicate definitions. The client code, C , is *not* in the scope of the predicates, so it can only use the predicates atomically and through the specifications of k_1, \dots, k_n . The predicate names can not occur in the conclusions specification, P and Q .

The side-conditions for this rule prevent both the predicates escaping the scope of the module, and repeated definitions of a predicate. The final restriction is not required but reduces the complexity of the modifies clauses for the frame rule.

In fact, the previous function definition rule is a derived rule in our system. It is derived from the standard function definition rule and two new rules for manipulating abstractions:

ABSTRACT WEAKENING

$$\frac{\Lambda; \Gamma \vdash \{P\}C\{Q\}}{\Lambda, \Lambda'; \Gamma \vdash \{P\}C\{Q\}}$$

where $\text{dom}(\Lambda')$ and $\text{dom}(\Lambda)$ are disjoint

ABSTRACT ELIMINATION

$$\frac{\Lambda, \Lambda'; \Gamma \vdash \{P\}C\{Q\}}{\Lambda; \Gamma \vdash \{P\}C\{Q\}}$$

where the predicate names in P, Q, Γ and Λ are not in $\text{dom}(\Lambda')$.

The first, ABSTRACT WEAKENING, allows the introduction of new definitions; and the second, ABSTRACT ELIMINATION allows any unused predicate to be removed.

We derive the abstraction function definition rule by taking the standard function definition rule, and using ABSTRACT WEAKENING on the client code premise and ABSTRACT ELIMINATION on the conclusion to remove the new predicate definitions. We can apply the same technique to the recursive function definition, however we do not require this for our examples.

Next we give one of the standard Hoare logic rules: the rule of consequence. (Of course, we use the other standard rules; space prevents us from listing them here.)

CONSEQUENCE

$$\frac{\Lambda \models P \Rightarrow P' \quad \Lambda; \Gamma \vdash \{P'\}C\{Q'\} \quad \Lambda \models Q' \Rightarrow Q}{\Lambda; \Gamma \vdash \{P\}C\{Q\}}$$

This rule is key to actual use of abstract predicate definitions. We provide the following two axioms concerning abstract predicates:

$$\begin{array}{ll} \text{OPEN} & (\alpha(\bar{x}) \stackrel{\text{def}}{=} P), \Lambda \models \alpha(\bar{E}) \Rightarrow P[\bar{E}/\bar{x}] \\ \text{CLOSE} & (\alpha(\bar{x}) \stackrel{\text{def}}{=} P), \Lambda \models P[\bar{E}/\bar{x}] \Rightarrow \alpha(\bar{E}) \end{array}$$

These axioms embody our intuition that if (and only if) an abstract predicate is in scope then we can freely move between its name and its definition.

Next we present the rules for function call and return.

$$\Lambda; \Gamma \vdash \{P[\bar{y}/\bar{x}]\} y=k(\bar{y}) \{Q[\bar{y}, y/\bar{x}, ret]\} \text{ where } \{P\}k(\bar{x})\{Q\} \in \Gamma$$

$$\Lambda; \Gamma \vdash \{P[x/ret]\} \text{return } x \{P\}$$

These rules use the distinguished variable *ret* to match the return value with its destination variable.

Finally we give the small axioms of separation logic

$$\Lambda; \Gamma \vdash \{E \mapsto _\} [E] = E' \{E \mapsto E'\}$$

$$\Lambda; \Gamma \vdash \{E \mapsto n \wedge x = m\} x=[E] \{E[m/x] \mapsto n \wedge x = n\}$$

$$\Lambda; \Gamma \vdash \{E \mapsto _\} \text{dispose}(E) \{\text{empty}\}$$

$$\Lambda; \Gamma \vdash \{\text{empty} \wedge x = m\} x=\text{cons}(\bar{E}) \{x \mapsto \bar{E}[m/x]\}$$

These refer only to the state that is accessed by the commands. They can typically be extended using the FRAME RULE to refer to a larger state, e.g.

$$\Lambda; \Gamma \vdash \{E \mapsto _\ast E_1 \mapsto E_2\} \text{dispose}(E) \{E_1 \mapsto E_2\}.$$

3.3 Examples

3.3.1 Connection pool

Our first example is a database connection pool. Constructing a database connection is generally an expensive operation, so this cost is reduced by pooling connections using the object pool design pattern [9]. Programs regularly access several different databases, hence we require multiple connection pools and dynamic instantiation (hence this could not be modelled in the framework of O’Hearn et al. [22]). The connection pool must prevent the connections being used after they are returned: ownership must be transferred between the client and the pool.

We assume a library routine, *consConn*, to construct a database connection. This routine takes a single parameter that specifies the database,⁴ and returns a handle to a connection. The specification uses a predicate *conn* to represent the state of the connection.

$$\{\text{empty}\} \text{ consConn}(s) \{\text{conn}(ret, s)\}$$

We define two abstract predicates for the connection pool module: *cpool* and *clist*. The *cpool* predicate is used to represent a connection pool; and the *clist* predicate is used inside the *cpool* to represent a list of connection predicates.

$$cpool(x, s) \stackrel{\text{def}}{=} \exists i. x \mapsto i, s * \text{clist}(i, s)$$

$$\text{clist}(x, s) \stackrel{\text{def}}{=} x \doteq \text{null} \vee (\exists i. x \mapsto i, j * \text{conn}(i, s) * \text{clist}(j, s))$$

where $E \doteq E'$ is a shorthand for $E = E' \wedge \text{empty}$.

The connection pool has three operations: construct a pool, *consPool*; get a connection, *getConn*; and free a connection, *freeConn*. These are specified as follows.

$$\begin{aligned} \{\text{empty}\} &\text{consPool}(s) \{cpool(ret, s)\} \\ \{cpool(x, s)\} &\text{getConn}(x) \{cpool(x, s) * \text{conn}(ret, s)\} \\ \{cpool(x, s) * \text{conn}(y, s)\} &\text{freeConn}(x, y) \{cpool(x, s)\} \end{aligned}$$

We give the implementation of these operations in Figure 2.

We present the proof that the *freeConn* implementation satisfies its specification, which illustrates the use of abstract predicates:

$$\begin{aligned} \{cpool(x, s) * \text{conn}(y, s)\} \\ \{\exists i. x \mapsto i, s * \text{clist}(i, s) * \text{conn}(y, s)\} \\ t=[x]; \end{aligned}$$

⁴In a more realistic implementation, such as JDBC [8], several arguments would be used to specify how to access a database.

```
let
  consPool s =
    (newvar p; p=cons(null,s); return p)
  getConn x = (newvar n,c,l,p; l=[x];
    if (l == null) then
      p=[x+1]; c=consConn(p)
    else (c=[l]; n=[l+1]; dispose(l);
      dispose(l+1); [x]=n);
    return c)
  freeConn x y =
    (newvar t,n; t=[x]; n=cons(y,t); [x]=n)
in
```

Figure 2: Source code for the connection pool

```
{x \mapsto t, s * clist(t, s) * conn(y, s)}
  n=cons(y, t);
{x \mapsto t, s * n \mapsto y, t * clist(t, s) * conn(y, s)}
  [x]=n
{x \mapsto n, s * n \mapsto y, t * clist(t, s) * conn(y, s)}
{x \mapsto n, s * clist(n, s)}
{cpool(x, s)}
```

In this proof the definitions of both *cpool* and *clist* are used with OPEN and CLOSE to give the following three implications

$$\begin{aligned} cpool(x, s) &\Rightarrow \exists i. x \mapsto i, s * \text{clist}(i, s) \\
n \mapsto y, t * \text{clist}(t, s) * \text{conn}(y, s) &\Rightarrow \text{clist}(n, s) \\
x \mapsto n, s * \text{clist}(n, s) &\Rightarrow cpool(x, s) \end{aligned}$$

These are used with the rule of CONSEQUENCE to complete the proof.

Next we present, and attempt to verify, a fragment of client code using the connection pool. It demonstrates both correct and incorrect usage, which causes the verification to fail. The example calls a function, *useConn*, that uses a connection.

```
{cpool(x, s)}
  y = getConn(x);
{cpool(x, s) * conn(y, s)}
  {conn(y, s)}
  useConn(y);
{conn(y, s)}
{cpool(x, s) * conn(y, s)}
  freeConn(x, Y);
{cpool(x, s)}
  useConn(y)
{???
```

The client gets a connection from the pool, uses it and then returns it. However, after returning it, the client tries to use the connection. This command cannot be validated as the precondition does not contain the *conn* predicate. Even though this predicate is contained in *cpool*, the client is unable to expand the definition because it is out of scope. This illustrates how abstract predicates capture “ownership transfer”. The connection passes from the client into the connection pool stopping the client from accessing it, even though the client has a pointer to the connection.

A connection pool library wants many instances; generally one per database. This can be easily handled by calling *consPool* the required number of times. Assume we have two different databases, *s1* and *s2*.

```
{empty}
  y = consPool(s1);
{conPool(y, s1)}
  z = consPool(s2);
{conPool(y, s1) * conPool(z, s2)}
```

This code creates two connection pools. The parameter prevents us returning the connection to the incorrect pool.

```

{conPool(y,s1) * conPool(z,s2)}
  x = getConn(z);
{conPool(y,s1) * conPool(z,s2) * conn(x,s2)}
  freeConn(y,x)
{???}

```

The `freeConn` call can only be validated if $s_1 = s_2$.⁵

This example has illustrated that abstract predicates capture the notion of “ownership transfer”, first presented with the hypothetical frame rule. Abstract predicates additionally deal with dynamic instantiation of a module, which the hypothetical frame rule cannot.

Note: To complete this example we should include a dispose pool function. As it presents no additional interesting difficulties we omit it from our exposition.

3.3.2 Malloc and free

The next example is a simple memory manager that allocates variable sized blocks of memory. We use a couple of additional features for handling arrays, described by Reynolds [29]: the iterated separating conjunctions, $\odot_{x=E_1}^{E_2}.P$; and a system routine `allocate` that allocates variable sized blocks. Intuitively the iterated separating conjunction, $\odot_{x=E_1}^{E_2}.P$, is the expansion

$$P[E_1/x] * \dots * P[E_2/x]$$

where x ranges from E_1 to E_2 . If E_2 is less than E_1 , it is equivalent to `empty`. More formally its semantics are:

$$\begin{aligned} S, H, I \models_{\Delta} \odot_{x=E_1}^{E_2}.P &\stackrel{\text{def}}{=} ([E_1]_{S,I} = n_1 \wedge [E_2]_{S,I} = n_2) \Rightarrow \\ &((n_1 \leq n_2 \Rightarrow S, H, I \models_{\Delta} P[n_1/x] * \odot_{x=n_1+1}^{n_2}.P) \\ &\quad \wedge (n_1 > n_2 \Rightarrow S, H, I \models_{\Delta} \text{empty})) \end{aligned}$$

Returning to the example, consider the following naïve specifications, which demonstrate the difficulties in reasoning about the memory manager:

```

{empty}malloc(n) {\odot_{i=0}^{n-1}.ret + i \mapsto _}
{\odot_{i=0}^{n-1}.x + i \mapsto _} free(x) {empty}

```

The problem is with the specification of `free`: it does not specify how much memory is returned as n is a free variable.

The standard specification [12] of `free` only requires it to deallocate blocks provided by `malloc`. Using abstract predicates we are able to provide an adequate specification.

```

{empty}malloc(n) {\odot_{i=0}^{n-1}.ret + i \mapsto _ * Block(ret, n)}
{\odot_{i=0}^{n-1}.x + i \mapsto _ * Block(x, n)} free(x) {empty}

```

The `Block` predicate is used as a modular certificate that `malloc` actually produced the block. The client can not construct a `Block` predicate as its definition is not in scope.

Standard implementations of `malloc` and `free` store the block’s size in the cell before the allocated block [12]. This can be specified by defining the `Block` predicate as follows.

$$\text{Block}(x, n) \stackrel{\text{def}}{=} x - 1 \mapsto n$$

This allows `free` to determine the quantity of memory returned.⁶

We can give a simple implementations of these routines that call system routines to construct (`allocate`) and dispose (`dispose`) the blocks.⁷

⁵Given the specification it is always valid to return a connection to a pool if it is to the correct database. A tighter specification could be given to restrict returning to the allocating pool.

⁶More complicated specifications can be used which account for padding and other book keeping.

⁷One could extend the specifications to have an additional memory manager predicate as in the connection pool example.

```

malloc n =(newvar x; x=allocate(n+1);
           [x]=n; return x+1)
free x =(newvar n; n=[x-1];
           while(n\geq 0) (n=n-1; dispose(x+n))

```

Both of their implementations can be verified; here we present the proof of `malloc`:

```

{empty}
  x=allocate(n+1);
{\odot_{i=0}^n.x + i \mapsto _}
{x \mapsto _ * \odot_{i=1}^n.x + i \mapsto _}
  [x]=n;
{x \mapsto n * \odot_{i=1}^n.x + i \mapsto _}
  return x+1
{ret - 1 \mapsto n * \odot_{i=1}^n.ret - 1 + i \mapsto _}
{ret - 1 \mapsto n * \odot_{i=0}^{n-1}.ret + i \mapsto _}
{\odot_{i=0}^{n-1}.ret + i \mapsto _ * Block(ret, n)}

```

The final implication in this proof abstracts the cell containing the block’s length, hence the client cannot directly access it. The following code fragment attempts to break this abstraction:

```

{empty}
  x=malloc(30);
{\odot_{i=0}^{29}.x + i \mapsto _ * Block(x, 30)}
  [x-1]=15;
{???}
  free(x);

```

The client attempts to modify the information about the block’s size. This would be a clear failure in modularity as the client is dependent on the implementation of `Block`. Fortunately, we are unable to validate the assignment as the pre-condition does not contain $x - 1 \mapsto _$. Although, the `Block` contains the cell, the client does not have the definition in scope and hence cannot use it.

O’Hearn, Reynolds and Yang’s [22] idealization of a memory manager does not support variable sized blocks. Their specifications can not be extended to cover this without exposing the representation of the block. Additionally, it is impossible for them to enforce that `malloc` must provide the blocks that `free` deallocates without extending the logic.

3.3.3 Permissions reading

O’Hearn [21] has recently given separation logic an ownership, or permissions, interpretation: $E \mapsto E'$ is the permission to read, write and dispose the cell at location E . Bornat et al. [5] extend this to allow read sharing. Essentially they annotate the \mapsto relation to express the type of permission it represents: read or total. In the previous example, the `Block` predicate is the permission to dispose the memory using `free`. Using this permissions reading of separation logic, abstract predicates allow modules to define their own permissions. The concept of ownership transfer can be seen as transferring permission to and from the client.

Consider a ticket machine:

```

{empty}getTicket() {Ticket(ret)}
{Ticket(x)}useTicket(x) {empty}

```

To call `useTicket` you must have called `getTicket`; each usage consumes a ticket. Trying to use a ticket twice fails:

```

{empty}
  x = getTicket();
{Ticket(x)}
  useTicket(x);
{empty}
  useTicket(x);
{???}

```

The second call to `useTicket` fails, because the first call removed the `Ticket`.

Any client that is validated against this specification must use the ticket discipline correctly. In fact the module is free to define the ticket in any way, e.g. $\text{Ticket}(x) \stackrel{\text{def}}{=} \text{true}$. Although this ticket would be logically valid to duplicate, $\text{true} * \text{true} \Leftrightarrow \text{true}$, the client does not know this, and hence cannot.

3.4 Semantics

In the previous section we have informally introduced the notion of abstract predicates and detailed a couple of examples to highlight their use and demonstrate their usefulness. In this section we formalize them precisely and show that the two abstract predicate rules are sound.

3.4.1 Programming language

We assume the usual semantics of separation logic [31] and extend it to handle the functions. A *semantic function environment*, Π , is a finite partial function from function names, k , to a pair of a vector of variable names and a command for the body ($\Pi : k \rightarrow (\overline{x}, C)$). An environment is well-formed, Π **ok**, if it only modifies local variables, $\forall \overline{x}, C \in \text{cod}(\Pi). \text{modifies}(C) = \emptyset$.

A configuration is defined as a quadruple of a function environment, a command, a stack, and a heap. A terminal configuration is a stack, heap pair or **Fault**. The semantics are given by a recursively defined relation between configurations and terminal configurations presented in Figure 3. We provide additional failure rules for each heap command accessing undefined state:

$$\left. \begin{array}{l} (\Pi, [E]=E', S, H) \Downarrow \text{Fault} \\ (\Pi, x=[E], S, H) \Downarrow \text{Fault} \\ (\Pi, \text{dispose}(E), S, H) \Downarrow \text{Fault} \end{array} \right\} \begin{array}{l} \text{where} \\ [E]_S \notin \text{dom}(H) \end{array}$$

and add rules to propagate the **Fault** states in the obvious way.

DEFINITION 3.1 (SAFETY).

$$(\Pi, C, S, H) : \text{safe} \stackrel{\text{def}}{=} \neg((\Pi, C, S, H) \Downarrow \text{Fault})$$

Note: As we only consider partial correctness, we consider non-termination as safe.

We have the standard properties required for the soundness of the frame rule [31].

LEMMA 3.2 (SAFETY MONOTONICITY).

$$(\Pi, C, S, H) : \text{safe} \wedge H' \perp H \Rightarrow (\Pi, C, S, H \circ H') : \text{safe}$$

LEMMA 3.3 (HEAP LOCALITY).

$$(\Pi, C, S, H_1) : \text{safe} \wedge (\Pi, C, S, H_1 * H) \Downarrow (S', H') \Rightarrow \exists H_2. H' = H * H_2 \wedge (\Pi, C, S, H_1) \Downarrow (S', H_2)$$

3.4.2 Abstract predicates

Next we define the semantics of an abstract predicate. First we define semantic predicate environments, Δ , as follows:

$$\Delta : \mathcal{A} \rightarrow \coprod_{n \in \mathbb{N}} (\mathbb{N}^n \rightarrow \mathcal{P}(\mathcal{H}))$$

where \mathcal{A} is the set of predicate names. We restrict our consideration to well-formed environments: each predicate name is mapped to a function of the correct arity, $\Delta(\alpha) : \mathbb{N}^{\text{arity}(\alpha)} \rightarrow \mathcal{P}(\mathcal{H})$. The reader might have expected the use of $\mathcal{P}(\mathcal{H} \times \mathcal{S} \times \mathcal{I})$, but this breaks substitution as the predicate can depend on variables that are not syntactically free.

The semantics of a predicate is as follows:

$$S, H, I \models_{\Delta} \alpha(\overline{E}) \Leftrightarrow \alpha \in \text{dom}(\Delta) \wedge H \in (\Delta\alpha)[[\overline{E}]_{S,I}]$$

The rest of the semantics are from the standard definition, sketched in §2, with the predicate environment added in the obvious way.

We define the following ordering on semantic predicate environments

$$\Delta \sqsubseteq \Delta' \stackrel{\text{def}}{=}$$

$$\forall \alpha. \forall \overline{n} : \mathbb{N}^{\text{arity}(\alpha)}. \Delta(\alpha) \neq \perp \Rightarrow \Delta(\alpha)(\overline{n}) \subseteq \Delta'(\alpha)(\overline{n})$$

The least upper bound of this order is written \sqcup .

LEMMA 3.4. *Well-formed semantic predicate environments form a complete lattice with respect to \sqsubseteq .*

LEMMA 3.5. *Formulae only depend on the predicate names they mention, i.e. if Δ defines all the predicate names in P , and Δ and Δ' are disjoint, then*

$$\forall S, H, I. \quad S, H, I \models_{\Delta} P \Leftrightarrow S, H, I \models_{\Delta \sqcup \Delta'} P$$

LEMMA 3.6. *Positive formulae are monotonic with respect to semantic predicate environments, i.e. if P is a positive formula,*

$$\Delta \sqsubseteq \Delta' \wedge S, H, I \models_{\Delta} P \Rightarrow S, H, I \models_{\Delta'} P$$

Now let us consider the construction of a semantic predicate environment from an abstract one, Λ . The abstract predicate environment does not, necessarily, define every predicate, so constructing a solution requires additional semantic definitions, Δ , to fill the holes. We use the following function to generate a fixed point:

$$\text{step}_{(\Delta, \Lambda)}(\Delta_n) \stackrel{\text{def}}{=} \lambda \alpha \in \text{dom}(\Lambda). \lambda \overline{n} \in \mathbb{N}^{\text{arity}(\alpha)}.$$

$$\{H | S, H, I \models_{\Delta_n \sqcup \Delta} \Lambda(\alpha)[\overline{n}]\}$$

where Λ are the definitions we want to solve; Δ are the predicates not defined in Λ ; and Δ_n is an approximation to the solution. step is monotonic on predicate environments, because of Lemma 3.6 and that all the definitions are positive. Hence by Tarski's theorem and Lemma 3.4 we know a least fixed point always exists. We write $[\Lambda]_{\Delta}$ for the least fixed point of $\text{step}_{\Delta, \Lambda}$.

Note: step is not Scott-continuous. This does not cause any problems because we only need consider the properties of the least fixed point, rather than its construction.

Consider the set of all solutions of Λ of the form $\Delta \sqcup [\Lambda]_{\Delta}$:

$$\text{close}(\Lambda) \stackrel{\text{def}}{=} \{\Delta \sqcup [\Lambda]_{\Delta} \mid \text{dom}(\Delta) = \mathcal{A} \setminus \text{dom}(\Lambda)\}$$

This function has two properties.

LEMMA 3.7. *Adding new predicate definitions refines the set of possible semantic predicate environments, i.e.*

$$\text{close}(\Lambda) \supseteq \text{close}(\Lambda, \Lambda')$$

LEMMA 3.8. *The removal of predicate definitions does not affect predicates that do not use them. Given Λ which is disjoint from Λ' and does not mention predicate names in its domain; we have*

$$\forall \Delta \in \text{close}(\Lambda). \exists \Delta' \in \text{close}(\Lambda, \Lambda'). \Delta \upharpoonright \text{dom}(\Lambda') = \Delta' \upharpoonright \text{dom}(\Lambda')$$

where $f \upharpoonright S$ is $\{a \mapsto b \mid a \mapsto b \in f \wedge a \notin \text{dom}(S)\}$

We define validity wrt an abstract predicate environment, written $\Lambda \models P$, as follows:

$$\forall S, H, I, \Delta \in \text{close}(\Lambda). \quad S, H, I \models_{\Delta} P$$

THEOREM 3.9. *OPEN and CLOSE, i.e.*

$$\begin{aligned} \alpha(\overline{x}) &\stackrel{\text{def}}{=} P, \Lambda \models \alpha(\overline{E}) \Rightarrow P[\overline{E}/\overline{x}] \\ \alpha(\overline{x}) &\stackrel{\text{def}}{=} P, \Lambda \models P[\overline{E}/\overline{x}] \Rightarrow \alpha(\overline{E}), \end{aligned}$$

are valid.

<i>Function definition</i>		<i>New variable</i>
$\frac{(\Pi[k_1 \mapsto (\overline{x_1}, C_1), \dots, k_n \mapsto (\overline{x_n}, C_n)], C, S, H) \Downarrow (S', H')}{(\Pi, \text{let } k_1 \overline{x_1} = C_1, \dots, k_n \overline{x_n} = C_n \text{ in } C), S, H) \Downarrow (S', H')}$		
<i>While1</i>		<i>Function call</i>
$\llbracket B \rrbracket_s = \text{false}$	<i>While2</i>	$\frac{(\Pi, C, \overline{x} \mapsto \llbracket \overline{y} \rrbracket_s, H) \Downarrow (S', H') \quad \Pi(k) = \overline{x}, C}{(\Pi, \text{newvar } x; C, S, H) \Downarrow (S_1[x \mapsto S(x)], H_1)}$
$(\Pi, \text{while } B C, S_1, H_1) \Downarrow (S_1, H_1)$		$\frac{\Pi(k) = \overline{x}, C}{(\Pi, x = k(\overline{y}), S, H) \Downarrow (S[x \mapsto \llbracket \text{ret} \rrbracket_{S'}], H')}$
<i>Sequence</i>		<i>If2</i>
$(\Pi, C_1, S_1, H_1) \Downarrow (S_2, H_2)$	<i>IfI</i>	$\frac{\llbracket B \rrbracket_s = \text{false} \quad (\Pi, C_2, S, H) \Downarrow (S_1, H_1)}{(\Pi, \text{if } B \text{ then } C_1 \text{ else } C_2, S, H) \Downarrow (S_1, H_1)}$
$(\Pi, C_2, S_2, H_2) \Downarrow (S_3, H_3)$		$\frac{\llbracket B \rrbracket_s = \text{true} \quad (\Pi, C_1, S, H) \Downarrow (S_1, H_1)}{(\Pi, \text{if } B \text{ then } C_1 \text{ else } C_2, S, H) \Downarrow (S_2, H_2)}$
$(\Pi, C_1; C_2, S_1, H_1) \Downarrow (S_3, H_3)$		
<i>Read</i>	$(\Pi, x = [E], S, H) \Downarrow (S[x \mapsto n], H)$	where $H(\llbracket E \rrbracket_s) = n$
<i>Write</i>	$(\Pi, [E] = E', S, H) \Downarrow (S, H[n \mapsto n'])$	where $\llbracket E \rrbracket_s = n, n \in \text{dom}(H)$ and $\llbracket E' \rrbracket_s = n'$
<i>Cons</i>	$(\Pi, x = \text{cons}(\overline{E}), S, H) \Downarrow (S[x \mapsto n], H[n \mapsto \overline{n}])$	where $\llbracket \overline{E} \rrbracket_s = n', \{n, \dots, n + n'\} \perp \text{dom}(H)$ and $\llbracket \overline{E} \rrbracket_s = \overline{n}$
<i>Dispose</i>	$(\Pi, \text{dispose}(E), S, H) \Downarrow (S, H')$	where $\llbracket E \rrbracket_s = n, H'[n \mapsto n'] = H$ and $n \notin \text{dom}(H')$
<i>Assign</i>	$(\Pi, x = E, S, H) \Downarrow (S[x \mapsto n], H)$	where $\llbracket E \rrbracket_s = n$
<i>Return</i>	$(\Pi, \text{return } E; , S, H) \Downarrow (S[\text{ret} \mapsto \llbracket E \rrbracket_s], H)$	

Figure 3: Operational semantics

3.4.3 Judgements

We are now in a position to define a semantics for our reasoning system. We write $\Lambda; \Gamma \models \{P\}C\{Q\}$ to mean that, if every specification in Γ is true of a function environment, and every abstract predicate definition in Λ is true of a predicate environment, then so is $\{P\}C\{Q\}$:

$$\Lambda; \Gamma \models \{P\}C\{Q\} \stackrel{\text{def}}{=} \forall \Delta \in \text{close}(\Lambda), \Pi. \quad \Pi \text{ ok} \wedge (\Delta \models_\Pi \Gamma) \Rightarrow \Delta \models_\Pi \{P\}C\{Q\}$$

where

$$\begin{aligned} \Delta \models_\Pi \Gamma &\stackrel{\text{def}}{=} \forall \{P\}k\{Q\} \in \Gamma. \Pi(k) = (\overline{x}, C) \Rightarrow \Delta \models_\Pi \{P\}C\{Q\} \\ \Delta \models_\Pi \{P\}C\{Q\} &\stackrel{\text{def}}{=} \forall S, H, I. S, H, I \models_\Delta P \Rightarrow ((\Pi, C, S, H) : \text{safe} \\ &\quad \wedge ((\Pi, C, S, H) \Downarrow (S', H') \Rightarrow S', H', I \models_\Delta Q)) \end{aligned}$$

Given this definition we can show that the two new rules for abstract predicates are sound.

THEOREM 3.10. *Abstract weakening is sound.*

PROOF. Direct consequence of definition of judgements and Lemma 3.7. \square

THEOREM 3.11. *Abstract elimination is sound.*

PROOF. Follows from Lemmas 3.5 and 3.8. \square

4. A JAVA-LIKE LANGUAGE

In the previous section we have shown how abstract predicates can be used with separation logic to provide a powerful but intuitive framework to reason about a language with first-order functions or procedures. We now turn our attention to another form of modularity: class-based objects.

More precisely we shall consider the problems of reasoning about a fragment of Java. We will consider a simple subset of Java based on Middleweight Java (MJ) [3]. We restrict MJ's expressions to be

Program	$\text{prog} ::= \text{cldef}_1 \dots \text{cldef}_n ; \overline{s}$
Class definition	$\text{cldef} ::= \text{class } C \text{ extends } D \{ \overline{\text{fdef}} \overline{\text{mdef}} \}$
Method definition	$\text{mdef} ::= C \text{ m}(C_1 x_1, \dots, C_n x_n) \{ \overline{s} \text{ return } x; \}$
Field definition	$\text{fdef} ::= C \text{ f};$
Expressions	$E ::= x \mid \text{null}$
Statements	$s ::= x = y.f; \mid x = (C)y; \mid x = \text{new } C();$ $\mid x.f = E; \mid x = y.m(\overline{E}); \mid C x;$ $\mid \{\overline{s}\} \mid ; \mid \text{if } (E == E) \{ \overline{s} \} \text{ else } \{ \overline{s} \}$

Figure 4: Syntax of MJ subset

stack variables and `null`,⁸ and remove constructors. We present the full syntax in Figure 4. We write `f` and `m` to range over field and method names respectively. We use `C`, `D` to range over class names, and `x`, `y` to range over variable names.

Consider giving a separation logic to this language: clearly we require the “points to” relation to describe fields.⁹ The new assertion “field points to”, written $x.f \mapsto y$, means the field f of the object x contains the value y . We also use the predicate $x : C$ to mean that x points to an object of class C : it is actually a C not just a subtype of C .

Now consider a motivational example [1] of a `Cell` class and a subclass that has backup, `Recell`, presented in Figure 5. The specifications of the `set` methods are:

$$\begin{array}{c|c} \{ \text{this}.cnts \mapsto _ \} & \{ \text{this}.cnts \mapsto X * \text{this}.bak \mapsto _ \} \\ \text{Cell.set}(n) & \text{Recell.set}(n) \\ \{ \text{this}.cnts \mapsto n \} & \{ \text{this}.cnts \mapsto n * \text{this}.bak \mapsto X \} \end{array}$$

These specifications have two problems: (a) they have no encapsulation; and (b) they do not respect behavioural subtyping.

(a) From the specification the client knows which field is used to

⁸This restriction is required as separation logic requires expressions to be pure: they cannot access the heap, i.e. $x.f_1.f_2$ is not allowed.

⁹An alternative approach would be to use the heap primitive as a whole object [18]. However, being able to split an object allows for more flexible reasoning.

```

class Cell {
    Object cnts;
    void set(Object n) {this.cnts = n;}
    Object get() {Object t;
        t = this.cnts; return t;}
}
class Recell extends Cell {
    Object bak;
    void set(Object n) {
        Object t; t = this.cnts;
        this.bak = t; this.cnts = n;}
}

```

Figure 5: Source code for Cell and Recell classes

store the contents. Clearly we need a greater level of abstraction. Using an abstract predicate allows us to encapsulate the object's state. We can write the `Cell`'s `set` specification as:

{ $Val_{Cell}(\text{this}, \cdot)$ } `Cell.set(n)` { $Val_{Cell}(\text{this}, n)$ }

and define the abstract predicate as

$$Val_{Cell}(\text{this}, x) \stackrel{\text{def}}{=} \text{this}.cnts \mapsto x$$

and scope it to the `Cell` class. This stops the `Cell`'s client using the field directly as it is hidden in the abstract predicate.

(b) Using standard behavioural subtyping [17], to allow dynamic dispatch, the `Recell`'s specification needs to be compatible with the `Cell`'s, i.e. we require the following two implications to hold

$$pre(\text{Cell}, \text{set}) \Rightarrow pre(\text{Recell}, \text{set}) \quad (1)$$

$$post(\text{Recell}, \text{set}) \Rightarrow post(\text{Cell}, \text{set}) \quad (2)$$

where $pre(C, m)$ denotes the pre-condition for the method m in class C , and $post$ denotes the post-condition.

Given the earlier specifications, these implications can never hold as they require a one element heap to be the same size as a two element heap. What about abstract predicates? The specification for `Recell` must use a different abstract predicate to `Cell` as it has a different body, i.e.

{ $Val_{Recell}(\text{this}, X, \cdot)$ } `Recell.set(n)` { $Val_{Recell}(\text{this}, n, X)$ } with the obvious definition for Val_{Recell} . Unfortunately the predicates are treated parametrically; no implications hold between them.

As it stands, abstract predicates do not, by themselves, help with behavioural subtyping. They provide support for encapsulation but not inheritance. In an object-oriented setting we require predicates to have multiple definitions, hence we introduce *abstract predicate families* where the families are sets of definitions indexed by class. Abstract predicate family instances¹⁰ are written $\alpha(x; \vec{v})$ to indicate that the object x satisfies *one* definition from the abstract predicate family α with arguments \vec{v} . The particular definition satisfied depends on the dynamic type of x . In object-oriented programming an object could be from one of many classes; abstract predicate families provide a similar choice of predicate definitions when considering their behaviour.

We define abstract predicate family definitions, Λ_f , with the following syntax.

$$\Lambda_f := \epsilon \mid \alpha_C \stackrel{\text{def}}{=} \lambda(x; \vec{x})P, \Lambda_f$$

Λ_f is well-formed if it has at most one entry for each predicate and class name pair, and the free variables of the body, P , are in its argument list, $x; \vec{x}$. We treat Λ_f as a function from predicate

¹⁰In the module system the concept of abstract predicate instance, and the abstract predicate are conflated, but here as we have multiple definitions the distinction must be kept.

and class name pairs to formulae. Each entry corresponds to the definition of an abstract predicate family for a particular class.

Our example shows the need to alter the arity of the predicate to reflect casting; the `Recell`'s predicate has three arguments while the `Cell`'s only has two. Hence we provide the following pair of implications:

$$\text{WIDEN} \quad \Lambda_f \models \alpha(x; \vec{x}) \Rightarrow \exists \vec{y}.\alpha(x; \vec{x}, \vec{y})$$

$$\text{NARROW} \quad \Lambda_f \models \exists \vec{y}.\alpha(x; \vec{x}, \vec{y}) \Rightarrow \alpha(x; \vec{x})$$

If we give a predicate more variables than its definition requires, it ignores them, and if too few, it treats the missing arguments as existentially quantified. This leads to our definition of substitution onto a predicate definition,

$$(\lambda(x; \vec{x}).P)[E; \vec{E}] \stackrel{\text{def}}{=}$$

$$\begin{cases} P[E/x, \vec{E}_1/\vec{x}] & |\vec{E}_1| = |\vec{x}| \text{ and } \vec{E} = \vec{E}_1, \vec{E}_2 \\ \exists \vec{y}.P[E/x, (\vec{E}, \vec{y})/\vec{x}] & |\vec{E}, \vec{y}| = |\vec{x}| \end{cases}$$

This definition of substitution can then be used to give the families' version of OPEN and CLOSE.

$$\text{OPEN} \quad \Lambda_f \models (x : C \wedge \alpha(x; \vec{x})) \Rightarrow \Lambda_f(\alpha, C)[x; \vec{x}]$$

$$\text{CLOSE} \quad \Lambda_f \models (x : C \wedge \Lambda_f(\alpha, C)[x; \vec{x}]) \Rightarrow \alpha(x; \vec{x})$$

where $\alpha, C \in \text{dom}(\Lambda_f)$.

To OPEN or CLOSE a predicate we must know which class contains the definition, and must have that version of the predicate in scope.

Note: We can OPEN predicates at incorrect arities as the substitution will correctly manipulate the arguments. An alternative approach would be to restrict opening to the correct arity, and use WIDEN and NARROW to get the correct arity. However, this approach complicates the semantics.

4.1 Proof rules

In this section we define a set of Hoare-style proof rules for reasoning about MJ programs. The judgements take the following form:

$$\Lambda_f; \Gamma \vdash \{P\}\vec{s}\{Q\}$$

where Γ is a set of assertions about methods. They have the following form:

$$\Gamma := \epsilon \mid \{P\}C.m(\vec{x})\{Q\}, \Gamma$$

A well-formed method environment, $\vdash \Gamma \text{ wf}$, defines each method and class name pair only once and has the following three properties:

1. The pre- and post-conditions can only contain free program variables in the argument list, `this` and `ret`; i.e.

$$\begin{aligned} \forall\{P\}C.m(\vec{x})\{Q\} \in \Gamma.\text{FPV}(P) &\subseteq (\{\text{this}\} \cup \vec{x}) \\ \text{FPV}(Q) &\subseteq (\{\text{this}, \text{ret}\} \cup \vec{x}) \end{aligned}$$

2. A method can only modify local variables; there are no global variables and arguments cannot be modified, i.e.

$$\forall\{P\}C.m(\vec{x})\{Q\} \in \Gamma.\text{modifies}(mbody(C, m)) = \emptyset$$

where $mbody(C, m)$ returns the body of method m in class C .

3. Subtypes must have compatible specifications with their supertypes, i.e.

$$\begin{aligned} \forall\{P_C\}C.m(\vec{x})\{Q_C\} \in \Gamma.D \prec C \\ \Rightarrow \{P_D\}D.m\{Q_D\} \in \Gamma \wedge \\ (\vdash \{P_C\}.\{Q_C\} \Rightarrow \{P_D\}.\{Q_D\}) \end{aligned}$$

DEFINITION 4.1 (SPECIFICATION COMPATIBILITY). We define specification compatibility, $\vdash \{P_C\}_{-\{Q_C\}} \Rightarrow \{P_D\}_{-\{Q_D\}}$, as $\forall \bar{s}. \text{if } \Lambda; \Gamma \vdash \{P_D\} \bar{s}\{Q_D\}$ is derivable from $\Lambda; \Gamma \vdash \{P_C\} \bar{s}\{Q_C\}$ using only the structural rules: CONSEQUENCE, AUXILIARY VARIABLE ELIMINATION and VARIABLE SUBSTITUTION.¹¹

This is more general than the behavioural subtyping rules as it allows manipulation of auxiliary variables. In fact, if the derivation only uses the rule of CONSEQUENCE, specification compatibility degenerates to behavioural subtyping.

Now let us consider the method call rule:

METHOD CALL

$$\Lambda; \Gamma \vdash \left\{ \begin{array}{l} P[x, \bar{y}/\text{this}, \bar{x}] \\ \wedge x \neq \text{null} \end{array} \right\} y=x.m(\bar{y})\{Q[x, y, \bar{y}/\text{this}, \text{ret}, \bar{x}]\}$$

where x has static type C and $\{P\}C.m(\bar{x})\{Q\} \in \Gamma$

The method call rule only needs to consider the static type of the receiver, because we have restricted ourselves to methods that are specification compatible.¹²

The rules for checking the whole program deserve some attention.

CLASS

$$\begin{aligned} & \Lambda_f; \Gamma \vdash \{P_n \wedge \text{this} : C\}mbody(C, m_n)\{Q_n\} \\ & \quad \vdots \\ & \Lambda_f; \Gamma \vdash \{P_1 \wedge \text{this} : C\}mbody(C, m_1)\{Q_1\} \\ & \Lambda_f; \Gamma \vdash \{P_1\}C.m_1(\bar{x}_1)\{Q_1\}, \dots, \{P_n\}C.m_n(\bar{x}_n)\{Q_n\} \\ \text{PROGRAM} \\ & \frac{\Lambda_{f_1}; \Gamma \vdash \Gamma_1 \dots \Lambda_{f_n}; \Gamma \vdash \Gamma_n \quad \emptyset, \Gamma \vdash \{P\} \bar{s}\{Q\}}{\vdash \{P\}cldef_1 \dots cldef_n; \bar{s}\{Q\}} \end{aligned}$$

where Γ_1 is the method specifications of the methods defined in and inherited into $cldef_1; \dots$; Γ_n is induced by $cldef_n$; $\Gamma = \Gamma_1, \dots, \Gamma_n$; and $\Lambda_{f_1}, \dots, \Lambda_{f_n}$ have disjoint domains.

These two rules correspond to the abstract function definition from the previous section. They enforce that each method is checked with the predicate definitions associated to its class.¹³ Inherited methods must be rechecked with the new predicate definitions for the class that inherits them. This is because when we check the method bodies in the *class* rule, we add to the pre-condition $\text{this} : C$. Without this we would not be able to open or close the abstract predicate families.

Again we have the two rules for introducing and eliminating abstract predicate families.

ABSTRACT WEAKENING

$$\frac{\Lambda_f; \Gamma \vdash \{P\}C\{Q\}}{\Lambda_f, \Lambda'_f; \Gamma \vdash \{P\}C\{Q\}}$$

where $dom(\Lambda'_f)$ and $dom(\Lambda)$ are disjoint.

ABSTRACT ELIMINATION

$$\frac{\Lambda_f, \Lambda'_f; \Gamma \vdash \{P\}C\{Q\}}{\Lambda_f; \Gamma \vdash \{P\}C\{Q\}}$$

where the predicates names in P, Q, Γ and Λ are not in $dom_a(\Lambda'_f)$ and define $dom_a(\Lambda_f)$ as $\{\alpha | (\alpha, C) \in dom(\Lambda_f)\}$.

¹¹The frame rule could be included in this list if \bar{s} is restricted to terms that modify no variables.

¹²We could present additional rules that do not rely on the subtyping constraint, but they would only serve to complicate the presentation and wouldn't illustrate anything interesting.

¹³We assume these definitions will be provided during the proof, and provide no explicit syntax for them.

Abstract predicate families are less symmetric than abstract predicates: weakening allows the introduction for a particular class and predicate, while elimination requires the entire family of definitions to be removed, i.e. must remove all the classes' definitions for a predicate. This is because it is not possible to give a simple syntactic check for which parts, i.e. classes, of a family will be used.

Finally we give the rules for field access, field write, and object construction, which are similar to their equivalents in the module system:

$$\begin{aligned} & \Lambda_f; \Gamma \vdash \{x.f \mapsto _\} x.f = E' \{x.f \mapsto E'\} \\ & \Lambda_f; \Gamma \vdash \left\{ \begin{array}{l} y.f \mapsto n \\ \wedge x = m \end{array} \right\} x = y.f \left\{ \begin{array}{l} y[m/x].f \mapsto n \\ \wedge x = n \end{array} \right\} \\ & \Lambda_f; \Gamma \vdash \{\text{empty}\} x = \text{new } C() \{x.f_1 \mapsto _\dots \mapsto x.f_n \mapsto _\wedge x : C\} \\ & \quad \text{where } C \text{ has fields } f_1 \dots f_n \end{aligned}$$

4.2 Example: Cell/Recell

Let us return to our original motivating example. We define an abstract predicate family, Val , with the definitions for `Cell` and `Recell` given earlier.

We have to validate four methods: `Cell.set`, `Cell.get`, `Recell.set` and `Recell.get`. Even though the bodies of `Cell.get` and `Recell.get` are the same, we must validate both, because they have different predicate definitions.

We give the proof for `Recell.set`.

$$\begin{aligned} & \{\text{Val}(\text{this}; X, _) \wedge \text{this} : \text{Recell}\} \\ & \{\text{this.cnts} \mapsto X * \text{this.bak} \mapsto _\wedge \text{this} : \text{Recell}\} \\ & \quad t = \text{this.cnts}; \\ & \{\text{this.cnts} \mapsto X * \text{this.bak} \mapsto _\wedge \text{this} : \text{Recell} \wedge X = t\} \\ & \quad \text{this.bak} = t; \\ & \{\text{this.cnts} \mapsto X * \text{this.bak} \mapsto t \wedge \text{this} : \text{Recell} \wedge X = t\} \\ & \quad \text{this.cnts} = n; \\ & \{\text{this.cnts} \mapsto n * \text{this.bak} \mapsto t \wedge \text{this} : \text{Recell} \wedge X = t\} \\ & \{\text{this.cnts} \mapsto n * \text{this.bak} \mapsto X \wedge \text{this} : \text{Recell}\} \\ & \{\text{Val}(\text{this}; n, X)\} \end{aligned}$$

The other method bodies are all easily verifiable.

Additionally, we must prove the method specifications are compatible. The compatibility of the `set` method follows from the rule of CONSEQUENCE and AUXILIARY VARIABLE ELIMINATION.

$$\frac{\vdash \{\text{Val}(\text{this}; X, _)\} \dots \{\text{Val}(\text{this}; n, X)\}}{\vdash \{\text{Val}(\text{this}; _, _)\} \dots \{\text{Val}(\text{this}; n, _)\}}$$

The `get` methods have the same specification, so are obviously compatible.

A client that uses this code does not need to worry about dynamic dispatch, because of the behavioural subtyping constraints. Consider the following method:

```
m(Cell c) {
    c.set(c);
}
```

This code simply sets the `Cell` to point to itself. The code is specified as

```
{Val(c, \_)} m(Cell c) ... {Val(c, c)}
```

Now consider calling `m` with a `Recell` argument.

```
{empty}
Recell r = new Recell(x);
{Val(r, x, \_)}
{Val(r, \_)}
m(r);
{Val(r, r)}
{Val(r, r, \_)}
```

We use CONSEQUENCE to cast the *Val* predicate to have the correct arity. We need not consider dynamic dispatch at all because of behavioural subtyping.

Note: The specification of method *m* is weaker than we might like. Based on the implementation we might expect the post-condition $\{ \text{Val}(r; r, x) \}$. However, there are several bodies that satisfy *m*'s specification: for example *c.set(x); c.set(c)*. We can set the *Cell* to have any value, as long as the last value we set is the *Cell* itself. This body acts identically on a *Cell* to the previous body, however on a *Recell* acts differently. Hence only using the specification we can not infer the tighter post-condition. This could be deduced if *m* was specified for a *Recell* as well.

4.3 Semantics

In this section we consider the extensions to the semantics of §3.4 sufficient to model abstract predicate families. MJ has been defined formally elsewhere [3]. First we shall make some small changes to the basics of the separation logic setting:

DEFINITION 4.2. A heap, H , is composed of two functions, $H = (H_v, H_t)$: the first, H_v , maps pairs of object identifiers and field names, (oid, f) , to values, val ; and the second, H_t , maps object identifiers to class names, C . We use $H(oid, f)$ to refer to the value given by the first function, $H_v(oid, f)$, and $H(oid)$ to refer to the value given by the second function, $H_t(oid)$.

(We make the obvious alterations to the semantics to deal with the new heap definition.) This definition allows a heap to contain only some fields of an object. This new definition also separates the type information from the value information in the heap.

We use the following two definitions to give the partial commutative heap composition monoid

$$(H'_v, H'_t) * (H''_v, H''_t) \stackrel{\text{def}}{=} (H'_v \circ H''_v, H'_t)$$

and is defined iff $\text{dom}(H'_v) \perp \text{dom}(H''_v)$ and $H'_t = H''_t$ where \circ is composition of disjoint partial functions.

The semantic predicate environment as defined in §3.4 has to be extended to handle the arity changes that predicate families require. We define a semantic predicate family environment as

$$\Delta_f : \mathcal{A} \times \mathbb{C} \rightarrow (\mathbb{N}^+ \rightarrow \mathbb{P}(\mathcal{H}))$$

This is a partial function from pairs of predicate and class name to semantic definition. An abstract predicate family is defined for all arities, so the semantic definition must be a function from *all* tuples of non-zero arity. This semantically supports the change in arity required by WIDEN and NARROW.

We can now give the semantics of the new assertions as follows

$$\begin{aligned} S, H, I \models_{\Delta_f} E.f \mapsto E' \\ \Leftrightarrow H(\llbracket E \rrbracket_{S,I}, f) = \llbracket E' \rrbracket_{S,I} \wedge \text{dom}(H) = \{ \llbracket E \rrbracket_{S,I}, f \} \\ S, H, I \models_{\Delta_f} E : C \Leftrightarrow H(\llbracket E \rrbracket_{S,I}) = C \\ S, H, I \models_{\Delta_f} \alpha(E; \bar{E}) \Leftrightarrow H \in (\Delta_f(\alpha, C))[\llbracket E \rrbracket_{S,I}; \bar{E}] \wedge H(\llbracket E \rrbracket_{S,I}) = C \end{aligned}$$

The field ‘‘points to’’ relation, $E.f \mapsto E'$, holds if the heap consists of a single field, f , of the object $\llbracket E \rrbracket_{S,I}$ and has the value $\llbracket E' \rrbracket_{S,I}$. $E : C$ is true if the heap types $\llbracket E \rrbracket_{S,I}$ as class C . $\alpha(E; \bar{E})$ holds for some heap H , where $\llbracket E \rrbracket_{S,I}$ has class C , iff H satisfies the predicate definition for C , given arguments $\llbracket E \rrbracket_{S,I}; \bar{E}$, in the predicate family α .

To ensure that WIDEN and NARROW hold we restrict our attention to argument refineable environments.

DEFINITION 4.3 (ARGUMENT REFINEABLE). A semantic predicate family environment is said to be argument refineable if adding

an argument can not increase, or ‘‘decrease’’, the set of accepting states, i.e.

$$AR(\Delta_f) \Leftrightarrow \forall \alpha, n, \bar{n}. \Delta_f(\alpha)[n; \bar{n}] = \bigcup_{n'} \Delta_f(\alpha)[n; \bar{n}, n']$$

PROPOSITION 4.4. Argument refinement coincides precisely with WIDEN and NARROW: $\forall S, H, I, \alpha, n, \bar{n}$.

$$AR(\Delta_f) \Leftrightarrow (S, H, I \models_{\Delta_f} \alpha(n; \bar{n}) \Leftrightarrow \exists n'. \alpha(n; \bar{n}, n'))$$

We can define an order on semantic predicate families environments, i.e

$$\Delta_f \sqsubseteq \Delta'_f \stackrel{\text{def}}{=} \forall \alpha, C, n, \bar{n}. \Delta_f(\alpha, C)[n; \bar{n}] \subseteq \Delta'_f(\alpha, C)[n; \bar{n}]$$

Again, the least upper bound of the order is written \sqcup . Lemmas 3.4 and 3.6 can be extended to semantic predicate family environments as follows:

LEMMA 4.5. Argument refineable predicate family environments form a complete lattice with respect to \sqsubseteq .

LEMMA 4.6. Positive formulae are monotonic with respect to semantic predicate family environments

$$\Delta_f \sqsubseteq \Delta'_f \wedge S, H, I \models_{\Delta_f} P \Rightarrow S, H, I \models_{\Delta'_f} P$$

However extending Lemma 3.5 is less straight forward, as it is not possible to tell which predicate name, class name pairs are used in a formula.

LEMMA 4.7. Formulae only depend on the abstractions they mention. If Δ_f contains all the abstractions in P , and $\text{dom}_a(\Delta_f) \cap \text{dom}_a(\Delta'_f) = \emptyset$, then

$$\forall S, H, I. S, H, I \models_{\Delta_f} P \Leftrightarrow S, H, I \models_{\Delta_f \sqcup \Delta'_f} P$$

Now let us consider the construction of semantic predicate family environments from their abstract syntactic counterparts. We define a new function, *stepf*, that accounts for the first argument's type and uses the special substitution,

$$\begin{aligned} \text{stepf}_{(\Lambda_f, \Delta_f)}(\Delta'_f)(\alpha, C)[n; \bar{n}] &\stackrel{\text{def}}{=} \\ \{ H | H(n) = C \wedge S, H, I \models_{\Delta_f \cup \Delta'_f} \Lambda_f(\alpha, C)[n; \bar{n}] \} \end{aligned}$$

This function is monotonic on predicate family environments, because of Lemma 4.6 and that all the predicate definitions are positive. Hence by Lemma 4.5 and Tarski’s theorem we know a fixed point must always exist. We write $\llbracket \Lambda_f \rrbracket_{\Delta_f}$ as the least fixed point of $\text{stepf}_{(\Lambda_f, \Delta_f)}$.

LEMMA 4.8. *stepf* produces argument refineable results.

Consider the following set of solutions:

$$\{ \llbracket \Lambda_f \rrbracket_{\Delta_f} \sqcup \Delta_f \mid (\mathcal{A} \times \mathbb{C}) \setminus \text{dom}(\Delta_f) = \text{dom}(\Lambda_f) \wedge AR(\Delta_f) \}$$

This satisfies the analogues of Lemmas 3.7 and 3.8.

LEMMA 4.9. Adding new predicate definitions refines the set of possible semantic predicate environments.

$$\text{close}(\Lambda_f) \supseteq \text{close}(\Lambda_f, \Lambda'_f)$$

LEMMA 4.10. The removal of predicate definitions does not affect predicates that do not use them, i.e. given Λ_f which is disjoint from Λ'_f and does not mention predicates in its domain; we have

$$\forall \Delta \in \text{close}(\Lambda_f). \exists \Delta'_f \in \text{close}(\Lambda_f, \Lambda'_f).$$

$$\Delta_f \upharpoonright \text{dom}(\Lambda'_f) = \Delta'_f \upharpoonright \text{dom}(\Lambda'_f)$$

where $f \upharpoonright S$ is $\{ a \mapsto b \mid a \mapsto b \in f \wedge a \notin \text{dom}(S) \}$

Validity is defined identically to the previous section, i.e.

$$\Lambda_f \models P \stackrel{\text{def}}{=} \forall S, H, I, \Delta_f \in \text{close}(\Lambda_f). S, H, I \models_{\Delta_f} P$$

THEOREM 4.11. OPEN and CLOSE, i.e.

$$\begin{aligned} \Lambda_f \models \alpha(E; \bar{E}) \wedge E : C \Rightarrow \Lambda_f(\alpha, C)[E, \bar{E}/x, \bar{x}] \\ \Lambda_f \models \Lambda_f(\alpha, C)[E, \bar{E}/x, \bar{x}] \wedge E : C \Rightarrow \alpha(E; \bar{E}) \end{aligned}$$

where $(\alpha, C) \in \text{dom}(\Lambda_f)$, are valid.

THEOREM 4.12. WIDEN and NARROW, i.e.

$$\begin{aligned} \Lambda_f \models \alpha(E; \bar{E}) \Rightarrow \exists X. \alpha(E; \bar{E}, X) \\ \Lambda_f \models \alpha(E; \bar{E}, E') \Rightarrow \alpha(E; \bar{E}), \end{aligned}$$

are valid.

4.3.1 Judgements

We are now in a position to define the semantics for our reasoning system. We write $\Lambda_f; \Gamma \models \{P\}C\{Q\}$ to mean if every specification in Γ is true of a method environment, and every abstract predicate family in Λ_f is true of a predicate family environment, then so is $\{P\}C\{Q\}$, i.e.

$$\begin{aligned} \Lambda_f; \Gamma \models \{P\}C\{Q\} &\stackrel{\text{def}}{=} \\ \forall \Delta_f \in \text{close}(\Lambda_f). (\Delta_f \models \Gamma) \Rightarrow \Delta_f \models \{P\}C\{Q\} \\ \text{where} \\ \Delta_f \models \Gamma &\stackrel{\text{def}}{=} \forall \{P\}C.m\{Q\} \in \Gamma. \Delta_f \models \{P\}m\text{body}(C, m)\{Q\} \\ \Delta_f \models \{P\}\bar{s}\{Q\} &\stackrel{\text{def}}{=} \\ \forall S, H, I, S, H, I \models_{\Delta_f} P &\Rightarrow ((S, H, \bar{s}, []): \text{safe} \\ \wedge ((S, H, \bar{s}, []) \rightarrow^* (S', H', v, [])) &\Rightarrow S', H', I \models_{\Delta_f} Q) \end{aligned}$$

Given this definition we can show that the two new rules for abstract predicate families are sound.

THEOREM 4.13. Abstract weakening is sound.

PROOF. Direct consequence of the definition of judgements and Lemma 4.9. \square

THEOREM 4.14. Abstract elimination is sound.

PROOF. Follows from Lemmas 4.7 and 4.10 \square

5. RELATED AND FUTURE WORK

In this paper we have considered the problem of writing specifications for programs that use various forms of abstraction. We have focused here on modules and Java-like classes. We have built on the formalism of separation logic and presented rules for reasoning about ADTs and Java-like classes. We have demonstrated the utility of these rules with a series of examples.

The principles of abstraction this paper builds on have been around since the Seventies. Parnas [25] first described the principles of information hiding and showed that without it seemingly independent components of a program could become tied together. Hoare provided a logic for data abstraction [11] that allowed internal implementation details to be hidden from the client. These ideas were developed further by Liskov [16] and Guttag [10] to provide what we now know as abstract datatypes.

In Hoare's [11] presentation of data abstraction, he used an abstraction function that maps values from a concrete domain to an abstract one. This abstraction function has been used in behavioural subtyping [17] to make classes with different implementations meet

the same specification. When reasoning with framing, Leino observed that, in addition to abstraction functions, datagroups [14] are needed to abstract *modifies*¹⁴ clauses. Abstract predicates, and families, combine the concept of both datagroups and the abstraction function into a single definition: separation logic formulae represent both the amount of state and its possible values.

There have been several attempts to reason about Java using a Hoare logic, including those by Oheimb and Nipkow [24], Poetzsch-Heffter and Müller [27], Pierik and de Boer [26]. However these logics do not have the framing properties of separation logic; method bodies must be verified at each call site. Also they do not attempt to express abstraction. In this sense Leino's work with datagroups [14] and data abstraction [15] are more closely related. This work uses the concept of “modular soundness” to determine when state can be exposed to a client. Another related approach by Barnett et al. [2] uses a private invariant to encapsulate the objects state. This invariant can be “packed” and “unpacked” to access its contents. These pack and unpack operations can be seen as corresponding to the open and close implications of abstract predicates.

Reddy [28] takes a different approach to adding abstraction to the logic. He extends specification logic to provide the ability to existentially quantify a predicate. This quantified predicate behaves in a similar way to an abstract predicate.

Middelkoop et al. [18] have similar aims to us and give a separation logic for a class-based language. Their approach considers an object as the primitive element of the heap, rather than a field. This restricts their use of the frame rule by preventing them from considering splitting an object. Their work does not consider abstraction or inheritance and so can not handle any of the examples presented in this paper.

A different approach to adding abstraction to separation logic has been taken by O'Hearn et al. [22]. They use the hypothetical frame rule to reason about static modularity. They are not able to reason about ADTs or classes, and cannot verify the examples we present. All the examples they present can be expressed using abstract predicates, however the proofs are less compact: predicates must be threaded through the proof to represent the internal invariant. This leads to two open questions: (1) can abstract predicates express all the proofs of the hypothetical frame rule?; and (2) can the concepts be soundly combined into a single logic? We believe the answer to both of these questions to be yes, but more work remains.

Building on the principles of the hypothetical frame rule, Mijajlović and Torp-Smith [19] have built a semantic model of refinement in a setting similar to separation logic. This allows them to semantically show one module could be used in place of another. They do not provide any logical rules for this reasoning, and they do not deal with ADTs. It would be interesting to see if their models could be adapted to abstract predicates.

A different approach to separation logic to dealing with the problem of aliasing is to impose some form of restriction using a type system. Ownership types [6] have been used to restrict aliasing in object-oriented languages. They prevent pointers into an object's representation, which helps reasoning about encapsulation. Smith and Drossopoulou [7] exploit this encapsulation to extend a Hoare logic with framing properties. Separation logic is more flexible than ownership types as it prevents dereferencing of a pointer rather than its existence.

Many researchers have pointed to similarities between separation logic and ownership types. However close comparison has been hampered by the fact that separation logic research has dealt with

¹⁴A *modifies* clause is an annotation that specifies all the possible changes made by a method/function body.

low-level pointer manipulation; whereas ownership types has dealt with high-level object languages. We hope that the work detailed in this paper may provide a stepping-stone for a more indepth analysis of these two approaches.

In this paper we have built a logic for reasoning about abstract types. It is well-known that abstract types correspond via the Curry-Howard correspondence to existential types [20]. Abstract predicates appear to be analogous, but at the level of the propositions themselves. We should also like to explore this analogy further, perhaps using higher-order logic to provide a logical semantics for abstract predicates.

The types analogy leads to another direction to pursue: parametric polymorphism. In this paper functions and methods can be defined to manipulate a datatype or class without knowing its representation: e.g. the connection pool did not know how a connection was stored. This is related to O’Hearn’s comment that “Ownership is in the eye of the asserter”. Abstract predicates may provide a suitable setting for studying parametric datatypes; we are currently working on a set of proof rules.

Finally, in this paper we have only considered sequential languages. Recently, O’Hearn has shown how to extend separation logic with rules to reason about concurrency primitives [21]. These rules use the same information hiding principles of the hypothetical frame rule [22]. They allow state to be stored in a semaphore, and by manipulating this semaphore the state can be transferred between threads. Unfortunately the semaphore is statically scoped, which prevents reasoning about heap allocated semaphores including, for example Java’s synchronised primitive. We are currently consider the combination of the information hiding provided by abstract predicates with O’Hearn’s system for concurrency to allow for reasoning about semaphores in the heap, and hence Java with threads.

Acknowledgements

We should like to thank Peter O’Hearn for insightful comments on earlier versions of this work and proposing the malloc and free example; and Andrew Pitts, Alasdair Wren and the anonymous referees for comments on this paper. We acknowledge funding from EPSRC (Parkinson) and APPSEM II (Bierman and Parkinson).

6. REFERENCES

- [1] M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1996.
- [2] M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 2004.
- [3] G.M. Bierman and M.J. Parkinson. Effects and effect inference for a core Java calculus. In *Proceedings of WOOD*, volume 82 of *ENTCS*, 2004.
- [4] L. Birkedal, N. Torp-Smith, and J.C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of POPL*, 2004.
- [5] R. Bornat, C. Calcagno, P. O’Hearn, and M. Parkinson. Permissions accounting in separation logic. *Proceedings of POPL*, 2005.
- [6] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of OOPSLA*, 2002.
- [7] S. Drossopoulou and M. Smith. Cheaper reasoning with ownership types. In *Proceedings of IWACO*, 2003.
- [8] J. Ellis and L. Ho. JDBC 3.0 specification, 2001.
<http://java.sun.com/products/jdbc/download.html>.
- [9] M. Grand. *Patterns in Java*, volume 1. Wiley, second edition, 2002.
- [10] J. Guttag. *The Specification and Applications to Programming of Abstract Data Types*. PhD thesis, Dept. of Computer Science, University of Toronto, 1975.
- [11] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
- [12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.
- [13] J. Lamping. Typing the specialization interface. In *Proceedings of OOPSLA*, 1993.
- [14] K.R.M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of OOPSLA*, 1998.
- [15] K.R.M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24:491–553, September 2002.
- [16] B. Liskov and S.N. Zilles. Programming with abstract data types. In *Proceedings of Symposium on Very High Level Programming Languages*, 1974.
- [17] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, 1994.
- [18] R. Middelkoop, K. Huizing, and R. Kuiper. A Separation Logic Proof System for a Class-based Language. In *Proceedings of LRPP*, 2004.
- [19] I. Mijajlović and N. Torp-Smith. Refinement in a separation context. In *Proceedings of FSTTCS*, 2004.
- [20] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.
- [21] P.W. O’Hearn. Resources, concurrency and local reasoning. Invited paper, in *Proceedings of CONCUR*, 2004.
- [22] P.W. O’Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. In *Proceedings of POPL*, 2004.
- [23] P.W. O’Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, 2001.
- [24] D. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In *Formal Methods Europe*, 2002.
- [25] D.L. Parnas. The secret history of information hiding. In *Software Pioneers: Contributions to Software Engineering*. Springer, 2002.
- [26] C. Pierik and F.S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In *Formal Methods for Open Object-Based Distributed Systems*, 2003.
- [27] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *Proceedings of ESOP*, 1999.
- [28] U.S. Reddy. Objects and classes in Algol-like languages. *Information and Computation*, 2002.
- [29] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, 2002.
- [30] R. Stata. Modularity in the presence of subclassing. Technical Report 145, Digital Equipment Corporation Systems Research Center, April 1997.
- [31] H. Yang. *Local reasoning for stateful programs*. PhD thesis, University of Illinois, July 2001.

Separation Logic, Abstraction and Inheritance

Matthew J. Parkinson

University of Cambridge, UK

Matthew.Parkinson@cl.cam.ac.uk

Gavin M. Bierman

Microsoft Research Cambridge, UK

gmb@microsoft.com

Abstract

Inheritance is a fundamental concept in object-oriented programming, allowing new classes to be defined in terms of old classes. When used with care, inheritance is an essential tool for object-oriented programmers. Thus, for those interested in developing formal verification techniques, the treatment of inheritance is of paramount importance. Unfortunately, inheritance comes in a number of guises, all requiring subtle techniques.

To address these subtleties, most existing verification methodologies typically adopt one of two restrictions to handle inheritance: either (1) they prevent a derived class from restricting the behaviour of its base class (typically by syntactic means) to trivialize the proof obligations; or (2) they allow a derived class to restrict the behaviour of its base class, but require that every inherited method must be reverified. Unfortunately, this means that typical inheritance-rich code either cannot be verified or results in an unreasonable number of proof obligations.

In this paper, we develop a separation logic for a core object-oriented language. It allows derived classes which override the behaviour of their base class, yet supports the inheritance of methods without reverification where this is safe. For each method, we require two specifications: a *static* specification that is used to verify the implementation and direct method calls (in Java this would be with a super call); and a *dynamic* specification that is used for calls that are dynamically dispatched; along with a simple relationship between the two specifications. Only the dynamic specification is involved with behavioural subtyping. This simple separation of concerns leads to a powerful system that supports all forms of inheritance with low proof-obligation overheads. We both formalize our methodology and demonstrate its power with a series of inheritance examples.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Program Verification—class invariants; D.3.3 [Programming Languages]: Language Constructs and Features—Classes and inheritance

General Terms Languages, Theory, Verification

Keywords Separation Logic, Modularity, Classes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'08, January 7–12, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-689-9/08/0001...\$5.00

1. Introduction

1.1 Motivation

Inheritance is a fundamental concept in object-oriented programming. It allows new classes to be defined in terms of existing classes. These new, or derived, classes inherit both attributes and behaviour from their base classes. There are several different uses of inheritance in object-oriented code:

Specialization: One common use of inheritance is to create a specialization of the base class. This typically involves directly inheriting members from the base class and extending this set with further members.

Overriding: Most object-oriented languages allow a class to replace some of the members of the base class, these members are said to override the definitions in the base class.

Code re-use: Sometimes inheritance is used purely for code reuse, that is, a derived class is not intended to be used in the same places as its base class, but rather they just share code.

Our main concern is providing practical, modular verification methodologies to enable programmers to reason about and document their code. As inheritance—in all its guises—plays such an important role in object-oriented code, we contend that any verification methodology must provide practicable techniques for dealing with it. In other words, the programmer should be able to verify common inheritance patterns without substantial rewriting of their code. Moreover, these typical patterns should not result in unreasonable proof obligations on the programmer.

Let us consider some examples of these uses of inheritance and the problems they raise for formal verification. In this paper we only address languages that support *single* inheritance; for example, languages such as C# and Java.¹ In Figure 1, we define a base class, Cell, and a derived class, Recell. This code uses both specialization and overriding. We will consider DCell and code re-use later. The base class has a field val, and two methods, set and get. The derived class directly inherits the field val, and method get. In addition it specializes its base class by defining a new field bak; and also overrides the set method. (The C# expression `base.m` accesses the `m` member of the base class of the current object. It is written `super.m` in Java.)²

Firstly, by defining the derived class, we are able to pass Recell objects as if they were Cell objects. This syntactic property is guaranteed by the type system. The corresponding semantic property, known as substitutivity, is that whenever an object of type Cell is

¹ In this paper, we shall write our code examples in a C#-style syntax, but our techniques apply to Java, Visual Basic and other single-inheritance object-oriented languages.

² The overridden set method is very abstract in using a `base` call for get, rather than direct field access or dynamic dispatch. We discuss the alternatives in §5.5.

```

class Cell
{
    public int val;

    public virtual void set(int x)
    {
        this.val=x;
    }

    public virtual int get()
    {
        return this.val;
    }
}

class Recell: Cell
{
    public int bak;
    public override void set(int x)
    {
        this.bak = base.get();
        base.set(x);
    }
}

class DCell: Cell
{
    public override void set(int x)
    {
        base.set(2*x);
    }
}

```

Figure 1. Examples of inheritance.

expected, supplying an object of type Recell will not change the behaviour of the program. Liskov and Wing (1994) defined a notion called *behavioural subtyping* that guarantees the property of substitutivity.

Secondly, Recell inherits the Cell's body for the get method. This is correct at the level of types, but is it semantically valid to inherit this method? Unfortunately, this is a non-trivial problem. To simplify matters, most current verification methodologies adopt one of two restrictions: either they (1) prevent a derived class restricting the behaviour of the base class which trivializes the proof obligations, e.g. Müller (2002); Barnett et al. (2004), or (2) they allow a derived class to restrict the behaviour of its base class, but require that all inherited methods are reverified (Parkinson and Bierman 2005). Neither of these approaches are satisfactory and one of the aims of this work was to remove these restrictions.

Now let us consider code reuse and the DCell class in Figure 1. As far as we are aware, most systems (for example, Barnett et al. (2005); Müller (2002)) cannot cope with this use of inheritance. The intention is that instances of class DCell always store double the value they have been set to. This use of inheritance is quite subtle; we have declared class DCell as a derived class essentially to enable us to inherit the get method code. However, it is clear that instances of DCell behave quite differently from instances of Cell, which is at odds with our assumptions of behavioural subtyping. (It's an instance of the "inheritance is not subtyping" phenomenon (Cook et al. 1990).)

Whilst such programming techniques are obviously fragile, it is our experience that this use of inheritance is quite common in the developer community. So, we aim to verify such uses of inheritance.

To summarize, our overall intention is to provide a flexible framework to allow programmers to formally specify and verify the behaviour of their object-oriented code. In this paper we concentrate specifically on verifying code that uses inheritance: where a class can re-use, extend or alter the representation and operations of its base class. We also impose a minimal set of requirements for any solution to this problem:

Soundness: We insist that our solution is sound, by which we mean that a verified program will satisfy its specification.

Modular: We insist that our solution is modular, by which we mean that when new components are added to the system, no old component needs to be re-specified or re-verified.

No base class code required: An inherited method need never be reverified, or, alternatively, a class need not see the code of its base class (Ruby and Leavens 2000).

Breadth: This is a harder criteria to quantify, but we insist that our approach can verify the typical patterns of inheritance use

in real-world software. In this case, we wish to support specialization, overriding and code re-use.

1.2 Our proposal

Our proposal for supporting inheritance builds on our earlier work (Parkinson and Bierman 2005) that developed a separation logic for reasoning about object-oriented code. Separation logic offers a particularly good framework as it supports local reasoning about stateful computation (Reynolds 2002) and hence deals directly with issues of ownership and state modification (in other systems these require additional complications to the underlying framework).

In earlier work (Parkinson and Bierman 2005) we proposed the notion of *abstract predicate families* to deal with the simple case of inheritance where every method is either overridden, or reverified in the derived class. Previously, when we verified a method body \bar{s} for class C with pre-condition P and post-condition Q , we verified the following.

$$\{P \wedge \text{this}:C\} \bar{s} \{Q\}$$

The type information, $\text{this}:C$, enables the use of abstract predicate families, but prevents inheritance of methods without reverification: the verification is specific to a single class.

In this work, we provide a generalized logic that allows these restrictions to be lifted. For each method, we require two specifications: a *static* specification, that is used to verify the implementation and direct method calls (in Java this would be with a super call); and a *dynamic* specification, that is used for calls that are dynamically dispatched; along with a simple relationship between the two specifications. Only the dynamic specification is involved with behavioural subtyping.

We will demonstrate the use of dynamic and static specifications by example, but first we must recap some details of abstract predicates and abstract predicate families. An abstract predicate has a name, a definition, and a scope. Within the scope one can freely swap between using the abstract predicate's name and its definition, but outside its scope it must be handled atomically, i.e. by its name. Thus the scope defines the abstraction boundary for the abstract predicate. Whilst this handles simple modules, it is not powerful enough to deal with object-oriented abstraction, as we wish each class to be able to provide its own definition of the predicate.

To deal with this, we introduced the notion of an abstract predicate family. Informally, it can be seen as a *dynamically dispatched* predicate. Just as the code for a method invocation is chosen based on the dynamic type of the instance parameter, we mirror this dispatch behaviour in the logic. An abstract predicate family uses its first argument to choose the definition of the predicate, i.e. if the first argument is of type C then the definition of the abstract predicate family is the one that class C defines:

$$x : C \Rightarrow (\alpha(x, \bar{y}) \Leftrightarrow \alpha_C(x, \bar{y}))$$

where α is an abstract predicate family, and α_C is the definition for class C.

Returning to our Cell example, we could define an abstract predicate family $Val(x, v)$, which for the Cell class is defined as $x.\text{val} \mapsto v$, that is when verifying the Cell class we assume:

$$x : \text{Cell} \Rightarrow (Val(x, v) \Leftrightarrow x.\text{val} \mapsto v)$$

Other classes are free to define their own entry for the *Val* family. This only specifies the definition for the Cell class; all other classes are unspecified, hence the proof is independent of their definition.

Now let us specify the Cell class. We will give the dynamic specification using abstract predicate families to allow more behavioural subtypes, and the static specification will closely mirror the actual implementation:

```
class Cell
```

```

{
    public int val;

    public virtual void set(int x)
    dynamic { Val(this,_) }-{ Val(this,x) }
    static { this.val ↠ _ }-{ this.val ↠ x )
    { ... }

    public virtual int get()
    dynamic { Val(this,x) }-{ Val(this,x) * ret = x }
    static { this.val ↠ x }-{ this.val ↠ x * ret = x }
    { ... }
}

```

The static specifications describe precisely how the methods work, that is `set` modifies the `val` field to contain `x`; and `get` returns the value stored in the `val` field.³ The dynamic specification is given in terms of the `Val` abstract predicate family to enable derived classes to alter the behaviour.

If the dynamic specification was given in terms of the fields accessed then a derived class would not be able to extend or alter the behaviour in any way (Leino 1998). Similarly if the static specification was given in terms of the abstract predicate family, then the derived class would not be able to inherit the method without knowing the hidden representation (Parkinson and Bierman 2005). By providing both a static and dynamic specification, we can inherit methods without reverification, and this also allows derived classes to alter the representation and behaviour if they override methods.

Let us return to the derived class `DCell`. We wish to provide a specification that allows it to be considered a behavioural subtype of `Cell` in spite of its radically different behaviour. The `DCell` class must behave the same as its parent's dynamic specification. If we define the `Val` predicate family as `false` for the `DCell` then this proof obligation becomes trivial.

$$x : \text{DCell} \Rightarrow (\text{Val}(x, v) \Leftrightarrow \text{false})$$

`DCell` ensures no client will ever have a `Val` predicate for a `DCell`. Therefore, in the “`Val`-world”, `DCell` is not a subtype of `Cell` (that is, a variable of static type `Cell` that satisfies `Val` will not point to a `DCell` object).

```

class DCell : Cell{
    public override void set(int x)
    dynamic { Val(this,_) }-{ Val(this,x) }
        also { DVal(this,_) }-{ DVal(this,x * 2) }
    { ... }

    public inherit int get()
    dynamic { Val(this,v) }-{ Val(this,v) * ret = v }
        also { DVal(this,v) }-{ DVal(this,v) * ret = v }
    }
}

```

Here we use `also` to mean satisfies both specifications, and `inherit` to provide a new specification for an inherited method. We introduce a new predicate family `DVal`, which defines the `DCell`'s behaviour. We specify `DVal` as⁴

$$x : \text{DCell} \Rightarrow (DVal(x, v) \Leftrightarrow x.\text{val} \mapsto v)$$

³ We could make the static specification more abstract to prevent a derived class depending on the precise representations of its base class. We could introduce a new predicate `ValCell(x, v)` for the entry in the `Val` family:

$$x : \text{Cell} \Rightarrow (\text{Val}(x, v) \Leftrightarrow \text{Val}_{\text{Cell}}(x, v))$$

This prevents the derived class depending on the unknown definition of the `Cell` class. As, we do in the rest of the paper.

⁴ We could alternatively specify it as

$$x : \text{DCell} \Rightarrow (DVal(x, v) \Leftrightarrow \text{Val}_{\text{Cell}}(x, v))$$

to be abstract in the `Cell`'s representation.

The `DCell` does satisfy the specification of `Cell`, but only vacuously. However, clients do not need to know the specification is only vacuously satisfied. They will never be able to observe this.

1.3 Contributions and content

This paper contains a number of novel contributions to the field of object-oriented verification. We explore the power of our system by considering a number of examples that exhibit typical uses of inheritance. Many of these examples are not supported by existing techniques. (Some more direct comparisons are given in §6.)

More specifically, the main contributions of this work are as follows.

- The separation of method specifications into “static” and “dynamic” specifications,
- The formalization of the proof obligations resulting from this separation,
- An elegant generalization of the formalization of abstract predicate families based on higher-order separation logic, and
- A systematic exploration of the expressive power of our logic by considering a number of typical programs exploiting various aspects of inheritance.

The proof system defined in this paper is modular, and does not require a derived class to see the code of its base class to verify its method. It can support many uses of inheritance: where a derived class extends its base class, where it restricts the behaviour of its base class, where it changes the behaviour of its base class, and even where it changes the representation of its base class. No other proof system that we are aware of can handle all of these uses of inheritance.

The rest of the paper is structured as follows. In §2 we define a core object-oriented language with annotated method definitions. In §3 we define formally our proof system, based on separation logic. In §4 we show how to simplify the annotations. In §5 we verify a number of example uses of inheritance. We conclude in §6 by comparing our proof system to others. In Appendix A we give an overview of the semantics of our proof system.

2. A programming language with specifications

In this section we define formally both the core object-oriented language we verify and the associated annotations.

2.1 Syntax

Our core language is an extended, featherweight fragment of C[#] called FVC[#] (for Featherweight Verified C[#]), that is similar to various formalized fragments of Java (Flatt et al. 1997; Bierman et al. 2004). The main extension to C[#] is that we annotate method definitions with static and dynamic specifications. The syntax of FVC[#] class definitions, method definitions, and statements is defined as follows.

FVC[#] programs

L ::= class C: D{ public \overline{T} \overline{f} ; \overline{A} \overline{K} \overline{M} } Class definitions

A ::= define $\alpha_C(\overline{x})$ as P Predicate Family Entry

K ::= public C() Sd Ss { \bar{s} } Constructor

M ::= Method definition

public virtual C m(\overline{D} \overline{x}) Sd Ss B Virtual method

public override C m(\overline{D} \overline{x}) Sd Ss B Overridden method

public inherit C m(\overline{D} \overline{x}) Sd Ss; Inherited method

Sd ::= dynamic S Dynamic specification

$Ss ::= \mathbf{static} \ S$	Static specification
$S ::= \{ \mathcal{P} \} \cdot \{ Q \}$	Method specification
$S \ \mathbf{also} \ \{ \mathcal{P} \} \cdot \{ Q \}$	
$B ::= \{ \bar{C} \bar{x}; \bar{s} \mathbf{return} y; \}$	Method body
$s ::=$	Statement
$x = y;$	Assignment
$x = \mathbf{null};$	Initialization
$x = y.f;$	Field access
$x.f = y;$	Field update
$x = y.m(\bar{z});$	Dynamic method invocation
$x = y.C::m(\bar{z});$	Direct method invocation
$x = (C)y$	Cast
$\mathbf{if}(x == y)\{s\} \ \mathbf{else} \ \{t\}$	Equality test
$x = \mathbf{new} \ C();$	Object creation

In the syntax rules we assume a number of metavariables: f ranges over field names, C, D over class names, m over method names, and x, y, z over program variables. We assume that the set of program variables includes a designated variable **this**, which cannot be used as an argument to a method (this restriction is imposed by the typing rules). We follow Featherweight Java, or FJ (Igarashi et al. 2001), and use an ‘overbar’ notation to denote sequences.

As with FJ, for simplicity we do not include any primitive types in FVC^\sharp , and we assume that there is a distinguished class Object that is at the root of the inheritance hierarchy. We do not formalize the type system of FVC^\sharp here as it is entirely standard.

A FVC^\sharp class definition, L , contains a collection of fields and method definitions and, for simplicity, a single constructor. A field is defined by a type and a name. A virtual method definition, M , is defined by a return type, a method name, an ordered list of arguments—where an argument is a variable name and a type—a method specification, S , and a method body, B .

A method specification consists of a dynamic specification, Sd , and a static specification, Ss , each consisting of a sequence of pre- and post-conditions separated by **also**. The use of these was informally presented in the previous section, and the formal conditions on their use is given later. In §4 we show how one can drop one or the other specification, but in our featherweight language we insist on both specifications to help in the definitions.

We also insist that all inherited methods are explicitly specified in the derived class. This is partly to simplify some definitions, but also to allow derived classes to provide new specifications for inherited methods. Clearly, outside the formalization we would not insist on specifying inherited methods—in which case it would be assumed that the method specifications were inherited also.

A method body, B , consists of a number of local variable declarations, followed by a sequence of statements and a **return** statement. The real economy of FVC^\sharp is that we do not have any syntactic forms for expressions (or even promotable expressions (Bierman et al. 2004)), and that the forms for statements are syntactically restricted. All expression forms appear only on the right-hand side of assignments. Moreover expressions only ever involve variables. In this respect, our form for statements is reminiscent of the A-normal form for λ -terms (Flanagan et al. 1993). A statement, s , is either an assignment, a field access, a field update, a method invocation, a cast, a conditional, or an object creation. In addition, we support direct method invocations using a C++-style syntax, e.g. $c.C::m(y)$. This syntax subsumes the standard C[♯] **base** calls: in a class derived from a base class B , the statement $x=\mathbf{base}.m(y)$ in C[♯] is written as $x=\mathbf{this}.B::m(y)$ in FVC^\sharp .

In FVC^\sharp we follow FJ and simplify matters by not considering overloading of methods or constructor methods. In spite of the

heavy syntactic restrictions, we have not lost any expressivity; it is quite simple to translate a more conventional calculus with expressions and promotable expressions into FVC^\sharp . Another advantage of our approach is that we have no need for the ‘stupid’ rules of FJ.

In FVC^\sharp we assume a rather large amount of syntactic regularity to make the definitions compact. All class definitions must (1) include a supertype; (2) start with all the declarations of the variables local to the method (hence a method block is a sequence of local variable declarations, followed by a sequence of statements); (3) have a **return** statement at the end of every method; and (4) write out field accesses explicitly, even when the receiver is **this**.

2.2 Dynamic semantics

The dynamic semantics of FVC^\sharp are routine and are omitted. However, it is interesting to compare the reduction rules for the two forms of method invocations. These are as follows.

$$\begin{array}{c} mbody(C, m) = (\bar{z}'', B) \\ \bar{B} \equiv \bar{C} \bar{x}; \bar{s}' \mathbf{return} x'; \quad \theta = [y_1, \bar{z}', \bar{x}' / \mathbf{this}, \bar{z}'', \bar{x}] \\ \bar{z}', \bar{x}' \text{ fresh} \quad S' = S[\bar{z}' \mapsto S(\bar{z})] \\ \hline S, H, y_0 = y_1.C::m(\bar{z}); \bar{s} \longrightarrow S', H, (\theta \bar{s}') y_0 = (\theta x'); \bar{s} \\ type(H(S(y))) = C \quad S, H, x = y.C::m(\bar{z}); \bar{s} \longrightarrow S', H', \bar{s}' \\ \hline S, H, x = y.m(\bar{z}); \bar{s} \longrightarrow S', H', \bar{s}' \end{array}$$

We define the dynamic semantics in terms of transitions between configurations. A configuration is a triple S, H, \bar{s} consisting of (1) a stack S , which is a map from identifiers to heap addresses; (2) a heap H , which is a map from heap addresses to object representations, where an object representation has a type and a map from field name to addresses; and (3) a sequence of statements, \bar{s} , which is the program being evaluated.

The interesting feature of the (dynamic dispatch) method invocation transition rule is its use of a direct method invocation.

3 Formalizing the proof system

In this section we formalize the proof system for reasoning about FVC^\sharp programs. An overview of the semantics of this system is given in Appendix A.

3.1 Logic syntax

In this subsection we define the fragment of separation logic that we use to reason about our FVC^\sharp programs. Formulae, \mathcal{P} are defined by the following grammar, where x, y, z ranges over variable names, α ranges over predicates. We encode the rest of the usual connectives.

Formulae

$$\begin{array}{l} P, Q ::= \forall x.P \mid P \Rightarrow Q \mid \mathbf{false} \mid \alpha(\bar{x}) \mid e = e' \mid x : C \\ \quad \mid x.f \mapsto e \mid P * Q \mid P - * Q \end{array}$$

$$e ::= x \mid \mathbf{null}$$

Separation logic (Ishtiaq and O’Hearn 2001; O’Hearn et al. 2001; Reynolds 2002) is an extension to Hoare logic that permits reasoning about shared mutable state. It extends Hoare logic by adding spatial connectives to the assertion language, which allow us to assert that two portions of the heap are disjoint, that is $P * Q$ means the heap can be split into two disjoint parts in which P and Q hold respectively. Space prevents us from giving a more thorough introduction; the reader is referred to the tutorial by Reynolds (2002) for a general introduction, and to Parkinson (2005) for an introduction to the use of separation logic for verifying object-oriented programs.

We define an environment, Γ , that contains the static and dynamic specifications specified in a FVC[#] program. We write $C.m(\bar{x}) : \{P\}.\{Q\} \in \Gamma$ to denote that Γ contains the dynamic specification $\{P\}.\{Q\}$ that is associated with the method m with parameters \bar{x} defined in class C . Similarly $C::m(\bar{x}) : \{S\}.\{T\} \in \Gamma$ denotes that Γ contains the static specification $\{S\}.\{T\}$ that is associated with the method m with parameters \bar{x} defined in class C . We assume two special specifications $C..ctor()$ and $C::..ctor()$ for the static and dynamic specifications of the constructor of class C .

We also need to define an environment, Δ , that stores the abstract predicate families and their definitions. In previous work (Parkinson and Bierman 2005), we defined this as a set of predicate definitions, along with some complicated rules for controlling its use. One contribution of this paper is to utilize some observations originating from more recent work on higher-order separation logic (Biering et al. 2007), to simplify this representation. We give the details in §3.2, but for now the reader can just consider Δ as containing the abstract predicate families and their definitions.

The judgements for reasoning about FVC[#] programs are of the form $\Delta; \Gamma \vdash \{P\}\bar{s}\{Q\}$, meaning that given environments Γ and Δ , the statement sequence \bar{s} satisfies the specification $\{P\}.\{Q\}$.

The axioms and rules for forming valid judgements can be divided into “structural rules” and “program rules”. The structural rules are those that work independently of the programs, i.e. they manipulate purely the pre- and post-conditions. These have been given elsewhere for separation logic (Parkinson 2005), so we do not repeat them here except for the “frame rule”, as it is crucial to the “local reasoning” principle of separation logic. It is defined as follows.

$$\frac{\Delta; \Gamma \vdash \{P\}\bar{s}\{Q\}}{\Delta; \Gamma \vdash \{P * R\}\bar{s}\{Q * R\}}$$

The program rules for forming valid judgements have also been given elsewhere. However, we give below the rules for both forms of method invocation. First, the rule for dynamic dispatch invocation, which is as follows.⁵

$$\frac{x \text{ has static type } C \quad C.m(\bar{x}) : \{P\}.\{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P[x, \bar{y}/\text{this}, \bar{x}] \wedge \text{this} \neq \text{null}\} \\ z = x.m(\bar{y}) \\ \{Q[z, x, \bar{y}/\text{ret}, \text{this}, \bar{x}]\}}$$

The rule for direct method calls is similar, except that the static specification is used.

$$\frac{C::m(\bar{x}) : \{S\}.\{T\} \in \Gamma}{\Delta; \Gamma \vdash \{S[x, \bar{y}/\text{this}, \bar{x}] \wedge \text{this} \neq \text{null}\} \\ z = x.C::m(\bar{y}) \\ \{T[z, x, \bar{y}/\text{ret}, \text{this}, \bar{x}]\}}$$

Finally, we give the rule for constructing a new object.

$$\frac{C..ctor() : \{P\}.\{Q\} \in \Gamma}{\Delta; \Gamma \vdash \{P\}x = \text{new } C()\{Q[\bar{x}/\text{this}]\}}$$

3.2 Abstract predicate families

Next we explain how we represent abstract predicate families in this framework (Parkinson and Bierman 2005). Rather than presenting a syntax for a context and rules for manipulating and using this context, we simply define our context, Δ , as a conjunction of formulae from our logic:

$$\Delta ::= P \mid \Delta_1 \wedge \Delta_2 \mid \exists \alpha. \Delta$$

The existential is used to hide redundant definitions.

⁵To simplify the presentation, we assume that the return variable, z , is not an argument or the instance parameter (receiver). This restriction can be satisfied trivially by adding an additional assignment to a fresh variable z' before the rule and replacing the uses of z by z' : $z' = z; z = x[z'/z].m(\bar{y}[z'/z])$

By using a formula from the logic the work can be connected with developments in higher-order separation logic (Biering et al. 2007; Nanevski et al. 2007; Krishnaswami et al. 2007). To simplify the presentation we do not use the full generality of higher-order separation logic in this paper: we simply require a second-order quantifier.

Now let us consider representing predicate definitions in this way. We might wish to define a predicate by **define** $Point(x, v)$ as $x.f \mapsto v$, this can be seen as saying the following formula is true:

$$\forall x, v. Point(x, v) \Leftrightarrow x.f \mapsto v$$

However, this is not powerful enough to reason about object-oriented abstractions. Due to dynamic dispatch, a method call is chosen based on the dynamic type of the instance parameter. Here we mirror this in the logic as the predicate definition is chosen by the dynamic type of the first parameter. So we could say **define** $Point_C(x, v)$ as $x.f \mapsto v$, which would mean two things: (1) the entry for the abstract predicate family $Point$ is $Point_C$; and (2) this entry is defined as $x.f \mapsto v$. These statements can be provided by the following

$$\begin{aligned} & (\forall x, v. Point_C(x, v) \Leftrightarrow x.f \mapsto v) \\ & \wedge (\forall x, v : C \Rightarrow (Point_C(x, v) \Leftrightarrow Point(x, v))) \end{aligned}$$

As a naming convention, we will use names, α , without a subscript to represent an abstract predicate *family*, and with α_C to represent the *entry* for class C in family α .

The role of the predicate arguments can be seen as analogous to the use of model fields in other work (Leino and Müller 2006). Derived classes generally introduce more model fields, hence we wish to interpret a predicate at many arities. To encode model fields more directly we could pass a record rather than a list of arguments. The implication below would then correspond to width subtyping and forgotten fields are existentially quantified. Returning to our point example, we might wish to be able to forget the outer most argument as a base class does not use this parameter:

$$Point(x) \Leftrightarrow \exists v. Point(x, v)$$

We can now define the formal translation of a definition to a logical assumption. We break the translation into three parts: (1) family to entry, $FtoE(\alpha, C)$ is a formula that connects the family, α , with the entry α_C assuming the first argument is of type C ; (2) entry to definition, $EtoD(\text{define } \alpha_C(x, \bar{x}) \text{ as } P)$ is a formula that connects the entry α_C with the definition P ; and (3) changing arity, $A(\alpha; n)$ defines that family α can be given any arity less than n and the missing values are existentially quantified.

$$\begin{aligned} FtoE(\alpha, C) &\stackrel{\text{def}}{=} \forall x, \bar{x}. x : C \Rightarrow (\alpha(x, \bar{x}) \Leftrightarrow \alpha_C(x, \bar{x})) \\ EtoD(\text{define } \alpha_C(x, \bar{x}) \text{ as } P) &\stackrel{\text{def}}{=} \forall x, \bar{x}. \alpha_C(x, \bar{x}) \Leftrightarrow P \\ A(\alpha; n+1) &\stackrel{\text{def}}{=} A(\alpha; n) \wedge \\ &\quad \forall y_1, \dots, y_n. \alpha(y_1, \dots, y_n) \Leftrightarrow \exists z. \alpha(y_1, \dots, y_n, z) \\ A(\alpha; 0) &\stackrel{\text{def}}{=} \text{true} \end{aligned}$$

We can translate an entry definition as (1) the formula connecting the family with the entry; (2) the formula connecting the entry with the definition; and (3) the formula specifying that the arity can be reduced.

$$\begin{aligned} apf_C(\text{define } \alpha_C(x, \bar{x}) \text{ as } P) &\stackrel{\text{def}}{=} FtoE(\alpha, C) \wedge EtoD(\text{define } \alpha_C(x, \bar{x}) \text{ as } P) \wedge A(\alpha; |\bar{x}|) \\ apf(\text{class } C : D \{ \text{public } \bar{T} \bar{f}; A_1 \dots A_n \text{ K } \bar{M} \}) &\stackrel{\text{def}}{=} apf_C(A_1) \wedge \dots \wedge apf_C(A_n) \end{aligned}$$

We have two rules to reason about these assumptions. The first allows us to strengthen our assumptions. If we can prove our program with just the assumptions Δ , then we can prove it with the

weaker assumptions Δ' .

$$\frac{\Delta' \Rightarrow \Delta \quad \Delta; \Gamma \vdash \{P\}\bar{s}\{Q\}}{\Delta'; \Gamma \vdash \{P\}\bar{s}\{Q\}} \text{ P-Weak}$$

The second proof rule allows the removal of predicates: this is the second-order analogy of the logical/ghost/auxiliary variable elimination rule.

$$\frac{\alpha \notin \text{FP}(P, Q, \Gamma) \quad \Delta; \Gamma \vdash \{Q\}\bar{s}\{R\}}{(\exists \alpha. \Delta); \Gamma \vdash \{Q\}\bar{s}\{R\}} \text{ P-Elim}$$

where $\text{FP}(P, Q, \Gamma)$ is the set of free predicate names in P, Q and Γ .

Finally, the rule of consequence is modified to take account of the environment Δ .

$$\frac{\Delta \Rightarrow (P \Rightarrow P') \quad \Delta; \Gamma \vdash \{P'\}\bar{s}\{Q'\} \quad \Delta \Rightarrow (Q' \Rightarrow Q)}{\Delta; \Gamma \vdash \{P\}\bar{s}\{Q\}} \text{ P-Elim}$$

3.3 Refinement and behavioural subtyping

A number of authors have offered definitions of behavioural subtyping, but in this work we follow Leavens and Naumann (2006) and propose a formulation in terms of a natural refinement order on specifications. We say that a specification $\{P_2\}_-\{Q_2\}$ refines another specification $\{P_1\}_-\{Q_1\}$, if for all programs \bar{s} , if \bar{s} satisfies the latter specification, then it also satisfies the former.

We characterize⁶ specification refinement using the structural rules of Hoare and Separation logic (Consequence, Frame, Logical⁷ variable elimination); that is, there exists a proof of the form

$$\frac{\begin{array}{c} \Delta \vdash \{P_1\}_-\{Q_1\} \\ \vdots \\ \Delta \vdash \{P_2\}_-\{Q_2\} \end{array}}{\Delta \vdash \{P_2\}_-\{Q_2\}}$$

When such a proof exists, we write $\Delta \vdash \{P_1\}_-\{Q_1\} \Rightarrow \{P_2\}_-\{Q_2\}$. We often need to introduce some type information in specification refinement. To do so, we define

$$\frac{\Delta \vdash \{P_1\}_-\{Q_1\} \xrightarrow{\text{this: } C} \{P_2\}_-\{Q_2\}}{\Delta \vdash \{P_1\}_-\{Q_1\} \stackrel{\text{def}}{\Rightarrow} \{P_2 * \text{this: } C\}_-\{Q_2\}}$$

We often need to combine specifications. This is written as **also**, and is encoded as follows using logical (auxiliary) variables.

Definition 1. $\{P_1\}_-\{Q_1\}$ **also** $\{P_2\}_-\{Q_2\}$ is defined as $\{(P_1 \wedge X=1) \vee (P_2 \wedge X \neq 1)\}_-\{(Q_1 \wedge X=1) \vee (Q_2 \wedge X \neq 1)\}_-$

We omit the X to mean selecting a fresh variable. In our verification rules, we omit **also** by encoding the specifications into a single specification.

Lemma 2.

1. $\Delta \vdash (\{P_1\}_-\{Q_1\} \text{ also } \{P_2\}_-\{Q_2\}) \Rightarrow \{P_1\}_-\{Q_1\}$
2. $\Delta \vdash (\{P_1\}_-\{Q_1\} \text{ also } \{P_2\}_-\{Q_2\}) \Rightarrow \{P_2\}_-\{Q_2\}$

3.3.1 Method verification

There are three forms of method definitions in FVC[#]. A method can be defined (1) as **virtual**, if it is not defined in its base class; or (2) as **inherit**, if it is not defined in this class, but is defined in its base class; or (3) as **override**, if it is defined both by this class and its base class. For each of these types of method definitions, we will provide the appropriate verification rule. The judgement form is written as $\Delta; \Gamma \vdash M$ in E , which means informally that “given environments Γ and Δ , the method definition M in class

⁶We currently do not have an adaption completeness result for our proof system, and so we cannot assert whether our syntactic characterization of refinement is complete. Yang’s thesis provides an adaption completeness result for separation logic without object-oriented features (Yang 2001).

⁷Sometimes called auxiliary or ghost variables.

E can be verified to meet its specification.” (In what follows we write $C \prec_1 D$ when class D is the immediate base class of derived class C , i.e. **class** $C : D \{ \dots \}$). Additionally, to simplify the presentation, we assume that methods do not modify the variables containing the arguments. Methods can be trivially rewritten to this form.)

First, we define the rule for verifying a new **virtual** method.

$$\frac{\begin{array}{c} B = \{ \bar{G} \bar{y}; \bar{s} \text{return } z; \} \\ Sd = \text{dynamic } \{P_E\}_-\{Q_E\} \\ Ss = \text{static } \{S_E\}_-\{T_E\} \\ \Delta \vdash \{S_E\}_-\{T_E\} \xrightarrow{\text{this: } E} \{P_E\}_-\{Q_E\} \quad (\text{Dynamic dispatch}) \\ \Delta; \Gamma \vdash \{S_E\}\bar{s}\{T_E[z/\text{ret}]\} \quad (\text{Body Verification}) \end{array}}{\Delta; \Gamma \vdash \text{public virtual } C m(\bar{D} \bar{x}) Sd Ss B \text{ in } E}$$

In this case there are just two proof obligations: we must verify that (1) the method body meets its **static** specification, (Body Verification); and (2) using the dynamic specification is valid for dynamic dispatch, (Dynamic dispatch). This second proof obligation forces a relationship between the static and dynamic specifications of a method. It corresponds to showing that if the object has type E and the dynamic specification is satisfied by the client, then the method body will execute successfully. Notice that by using the static specification we do not have to verify the body against the dynamic specification.

Next, we define the verification rule for inheriting a method.

$$\frac{\begin{array}{c} E \prec_1 F \\ Sd = \text{dynamic } \{P_E\}_-\{Q_E\} \\ Ss = \text{static } \{S_E\}_-\{T_E\} \\ F.m(\bar{x}): \{P_F\}_-\{Q_F\} \in \Gamma \\ F::m(\bar{x}): \{S_F\}_-\{T_F\} \in \Gamma \\ \Delta \vdash \{P_E\}_-\{Q_E\} \Rightarrow \{P_F\}_-\{Q_F\} \quad (\text{Behavioural Subtyping}) \\ \Delta \vdash \{S_F\}_-\{T_F\} \Rightarrow \{S_E\}_-\{T_E\} \quad (\text{Inheritance}) \\ \Delta \vdash \{S_E\}_-\{T_E\} \xrightarrow{\text{this: } E} \{P_E\}_-\{Q_E\} \quad (\text{Dynamic dispatch}) \end{array}}{\Delta; \Gamma \vdash \text{public inherit } C m(\bar{D} \bar{x}) Sd Ss; \text{ in } E}$$

In this case there are three proof obligations: we must verify that (1) the new dynamic specification is a valid behavioural subtype, (Behavioural Subtyping); (2) the method meets the static specification, (Inheritance); and (3) (as before) using the dynamic specification is valid for dynamic dispatch, (Dynamic dispatch). The first proof obligation amounts to requiring that whenever it is valid to use the dynamic specification of the base class, it is also valid to use the dynamic specification of this class, E . The second proof obligation amounts to showing that the inherited method body satisfies the new static specification. However, this rule does *not* use the inherited method body at all; it is *not* needed. The rule works purely at the level of the specifications.

Finally, we give the verification rule for overriding a method.

$$\frac{\begin{array}{c} E \prec_1 F \\ F.m(\bar{x}): \{P_F\}_-\{Q_F\} \in \Gamma \\ B = \{ \bar{G} \bar{y}; \bar{s} \text{return } z; \} \\ Sd = \text{dynamic } \{P_E\}_-\{Q_E\} \\ Ss = \text{static } \{S_E\}_-\{T_E\} \\ \Delta \vdash \{P_E\}_-\{Q_E\} \Rightarrow \{P_F\}_-\{Q_F\} \quad (\text{Behavioural Subtyping}) \\ \Delta \vdash \{S_E\}_-\{T_E\} \xrightarrow{\text{this: } E} \{P_E\}_-\{Q_E\} \quad (\text{Dynamic dispatch}) \\ \Delta; \Gamma \vdash \{S_E\}\bar{s}\{T_E[z/\text{ret}]\} \quad (\text{Body Verification}) \end{array}}{\Delta; \Gamma \vdash \text{public override } C m(\bar{D} \bar{x}) Sd Ss B \text{ in } E}$$

Again there are three proof obligations: we must verify that: (1) the new dynamic specification is a valid behavioural subtype, (Behavioural Subtyping); (2) the method body meets the static specification, (Body Verification); and (3) using the dynamic specification is valid for dynamic dispatch, (Dynamic dispatch). This verification is almost identical to the previous, but here we can verify the body of the method against the static specification as it is defined in this class, E .

The second verification rule has a degenerate (but common) form, when a derived class inherits a method from a base class but does *not* provide a new specification. In this case, the verification rule degenerates to the following.

$$\frac{\begin{array}{l} E \prec_1 F \\ F.m(\bar{x}): \{P_F\} - \{Q_F\} \in \Gamma \\ F::m(\bar{x}): \{S_F\} - \{T_F\} \in \Gamma \\ \Delta \vdash \{S_F\} - \{T_F\} \xrightarrow{\text{this: } E} \{P_F\} - \{Q_F\} \quad (\text{Dynamic dispatch}) \end{array}}{\Delta; \Gamma \vdash \text{public inherit } C m(\bar{D} \bar{x}); \text{in } E}$$

There is just one proof obligation, which amounts to verifying that if the object has type E and the dynamic specification of the base class is satisfied by the client, then the method body will satisfy the specification. Again, it is worth pointing out that this proof obligation is at the level of the specifications; we do not need the inherited method body from the base class.

Finally, we need a special verification rule for a constructor method definition. This is as follows.

$$\frac{\begin{array}{l} E \prec_1 F \\ \text{fields}(E) = f_1, \dots, f_n \\ F::\text{ctor}(): \{S_F\} - \{T_F\} \in \Gamma \\ Sd = \text{dynamic } \{P_E\} - \{Q_E\} \\ Ss = \text{static } \{S_E\} - \{T_E\} \\ \Delta \vdash \{S_E\} - \{T_E\} \xrightarrow{\text{this: } E} \{P_E\} - \{Q_E\} \\ \Delta; \Gamma \vdash \{T_F * R * Fs\} \bar{s} \{T_E\} \quad (\text{Body Verification}) \\ S_E \Rightarrow S_F * \text{true} \\ R \Leftrightarrow S_F - \circledast S_E \\ Fs = \text{this}.f_1 \mapsto _ * \dots * \text{this}.f_n \mapsto _ \end{array}}{\Delta; \Gamma \vdash \text{public } E() Sd Ss \{S\}}$$

This rule is complicated by the implicit base call at the beginning of the constructor, that is, before the constructor body \bar{s} begins executing, the base class constructor is executed. When verifying the body (Body Verification) the pre-condition is composed of three things: (1) the post-condition of the base class constructor call, (2) a formula R which intuitively is the disjoint state required by the constructor (and hence is given by the formula $S_F - \circledast S_E$ ⁸), and (3) a representation of the fields defined in E but not including the fields inherited from the base class.

3.4 Class verification

We can now use the method verification rules given above to verify a definition. The rule is as follows.

$$\frac{}{\Delta; \Gamma \vdash \text{class } C: D \{ \text{public } \bar{T} \bar{f}; \bar{A} \bar{K} \bar{M} \}}$$

Informally, this means that to verify a class definition one must verify every method.

The rule for verifying a complete program is then as follows.

$$\frac{\begin{array}{l} \Gamma = \text{specs}(L_1 \dots L_n) \\ apf(L_1); \Gamma \vdash L_1 \dots \\ \dots \\ apf(L_n); \Gamma \vdash L_n \\ \text{true}; \Gamma \vdash \{\text{true}\} \bar{s} \{\text{true}\} \end{array}}{\vdash L_1 \dots L_n \bar{s}}$$

Informally, this rule states that under the assumption that all the specifications are correct, every class definition must be verified,⁹ along with verifying the main body, \bar{s} .

Properties Given the proof rules above, we can now reconsider the criteria given in §1.1. First, our proof system is *sound*.

Theorem 3. *The program verification rule is sound. (See Appendix A for a formal statement of soundness and an overview of the proof.)*

⁸ $P - \circledast Q$ is defined as $\neg(P - * - Q)$ and intuitively means subtracting P from Q .

⁹ This assumption is valid as we are only dealing with partial correctness.

Secondly, our system is *modular*, i.e. the introduction of new methods or classes does not invalidate an existing proof. Thirdly, each method body is verified only once, even when defining an overridden method or inheriting a method from the base class.

Finally, in §5 we consider the applicability of our system by considering a number of typical uses of inheritance in object-oriented code.

4. Simplifying annotations

In many cases the dynamic and static specifications turn out to be very similar. Fortunately, there is a relatively simple process to derive the static specification from the dynamic, and vice versa.

Deriving static from dynamic We give a syntactic ‘opening’ function, $\llbracket P \rrbracket_C$, which opens all the abstract predicate families in P on the object **this** at type C . That is,

$$\begin{aligned} \llbracket \alpha(\mathbf{this}, \bar{x}) \rrbracket_C &\stackrel{\text{def}}{=} \alpha_C(\mathbf{this}, \bar{x}) \\ \llbracket \alpha(y, \bar{x}) \rrbracket_C &\stackrel{\text{def}}{=} \alpha(y, \bar{x}) \quad \text{where } y \neq \mathbf{this} \\ \llbracket \alpha_D(\bar{x}) \rrbracket_C &\stackrel{\text{def}}{=} \alpha_D(\bar{x}) \\ \llbracket pr(\bar{x}) \rrbracket_C &\stackrel{\text{def}}{=} pr(\bar{x}) \\ \llbracket \text{false} \rrbracket_C &\stackrel{\text{def}}{=} \text{false} \\ \llbracket P op Q \rrbracket_C &\stackrel{\text{def}}{=} \llbracket P \rrbracket_C op \llbracket Q \rrbracket_C \quad \text{where } op ::= * | \Rightarrow | \neg \\ \llbracket \forall x. P \rrbracket_C &\stackrel{\text{def}}{=} \forall x. \llbracket P \rrbracket_C \end{aligned}$$

where $pr(x, y)$ is either $x.f \mapsto y$, $x = y$ and $x : C$.

Lemma 4. $\mathbf{this} : C \implies (P \Leftrightarrow \llbracket P \rrbracket_C)$

Hence, given a dynamic specification $\{P\} - \{Q\}$, we can derive the static specification $\{\llbracket P \rrbracket_C\} - \{\llbracket Q \rrbracket_C\}$, which automatically satisfies the (Dynamic dispatch) proof obligation.

Lemma 5. $\{\llbracket P \rrbracket_C\} - \{\llbracket Q \rrbracket_C\} \xrightarrow{\mathbf{this}: C} \{P\} - \{Q\}$

$$\frac{\text{Proof.} \quad \frac{\{\llbracket P \rrbracket_C * \mathbf{this}: C\} - \{\llbracket Q \rrbracket_C * \mathbf{this}: C\}}{\{P * \mathbf{this}: C\} - \{Q\}} \text{ Frame}}{\text{Conseq}} \quad \square$$

Both the static specifications of the set and get methods of Cell can be inferred in this way.

Deriving dynamic from static If we only provide a static specification for a method, we assume the dynamic specification is identical to the static specification. This also satisfies the (Dynamic dispatch) proof obligation trivially.

5. Examples

In this section we give a number of examples to demonstrate the power and applicability of our proof system. All our examples involve inheritance of the Cell class that we described in §1. For completeness we give the complete definition of the Cell class, including the abstract predicate families and method specifications, in Figure 2.

Before we consider inheriting this class, we should first verify that it meets its own specification! For the three virtual methods, this means the two proof obligations, (Body Verification) and (Dynamic dispatch). Luckily, for all three methods the latter proof obligation is satisfied following Lemma 5. We give below a verification of the set method body, and suppress the verifications of get and swap.

$$\begin{aligned} &\{\text{ValCell}(\mathbf{this}, _)\\ &\quad \{\mathbf{this}.val \mapsto _\} \mathbf{this}.val = x; \{\mathbf{this}.val \mapsto x\}\\ &\quad \{\text{ValCell}(\mathbf{this}, x)\} \end{aligned}$$

```

class Cell {
    int val;
    define ValCell(x, v) as x.val  $\mapsto$  v

    public Cell() dynamic {true} -{ Val(this, _) } {}

    public virtual void set(int x)
    dynamic { Val(this, _) } -{ Val(this, x) }
    { this.val=x; }

    public virtual int get()
    dynamic { Val(this, v) } -{ Val(this, v) * ret = v }
    { return this.val; }

    public virtual void swap(Cell c)
    static { Val(this, v1) * Val(c, v2) } -{ Val(this, v2) * Val(c, v1) }
    { int t,t2; t = this.get(); t2 = c.get(); this.set(t2); c.set(t); }
}

```

Figure 2. Source code for Cell examples

Before turning to deriving from this class, we consider in a little more detail the `swap` method. It is an example of the template method pattern (Gamma et al. 1994). It does not manipulate the data directly, but simply uses other methods to update the state. This allows the code to be reused in derived classes that alter the representation.

Hence it is interesting to consider the consequences of inheriting this method in some derived class, C, of `Cell`. If the derived class C inherits `swap` and does not alter its specification, then its only proof obligation is (Dynamic dispatch) which follows trivially (using the rule of consequence) as follows.

$$\frac{\{ Val(this, v_1) * Val(c, v_2) \} - \{ Val(this, v_2) * Val(c, v_1) \}}{\{ Val(this, v_1) * Val(c, v_2) * this:C \} - \{ Val(this, v_2) * Val(c, v_1) \}}$$

Thus, any derived class is essentially free to inherit this method. It is particularly interesting to explore the consequences of different implementations of the `swap` method and the effects on the ability of derived classes to inherit this method. For example, consider if we had implemented `swap` using direct field access on the object, e.g.

```

public virtual void swap1(Cell c)
dynamic { Val(this, v1) * Val(c, v2) } -{ Val(this, v2) * Val(c, v1) }
static { ValCell(this, v1) * Val(c, v2) } -{ ValCell(this, v2) * Val(c, v1) }
{ int tmp = c.get(); c.set(this.val); this.val = tmp; }

```

The proof obligation to inherit this method would impose a constraint on any derived class. A better alternative would probably be to override the method.

A more optimised implementation of `swap` could directly access the fields of `c` as well.

```

public virtual void swap2(Cell c)
static { Val(this, v1) * Val(c, v2) } -{ Val(this, v2) * Val(c, v1) }
{ int tmp = c.val; c.val = this.val; this.val = tmp; }

```

For the proof obligations to be satisfied, this would effectively impose a global constraint on all derived classes of `Cell`, that they preserve the usage of the `val` field.¹⁰

$$Val(x, v) \Leftrightarrow ValCell(x, v) * ValLeft(x) \quad (1)$$

This kind of constraint is analogous to the condition in other work (Müller 2002) that derived class invariants cannot restrict

¹⁰ Our program verification rules do not directly support this constraining of derived classes, but this could be trivially added.

```

class Recell : Cell{
    int bak;

    define ValRecell(x, v, o) as ValCell(x, v) * x.bak  $\mapsto$  o

    public Recell () dynamic {true} -{ Val(this, _) } {}

    public inherit int get()
    dynamic { Val(this, v, o) } -{ Val(this, v, o) * ret = v }

    public override void set(int x)
    dynamic { Val(this, v, _) } -{ Val(this, x, v) }
    { this.bak = this.Cell::get(); this.Cell::set(x); }

    public virtual void undo()
    dynamic { Val(this, v, o) } -{ Val(this, o, _) }
    { int tmp = this.bak; this.Cell::set(tmp); }
}

```

Figure 3. The Recell class

the base class invariant. The derived class must define a meaning for the predicate family `ValLeft`. As we see later, this kind of constraint prevents many useful subtypes.

5.1 Specialisation: Recell

Now let us consider `Recell`, a derived class of `Cell`, which additionally stores the previous value that was set to allow `undo`. The code is given in Figure 3. We consider the methods in turn: First, let us consider the `get` method. We must show that it is valid to inherit this method into the `Recell` class. First, we need to verify the proof obligation (Inheritance) i.e. to show that the static specification of the method in `Recell` refines the `Cell` static specification. This can be proved as follows.

$$\frac{\frac{\{ ValCell(x, v) \} - \{ ValCell(x, v) * ret=v \}}{\{ ValCell(x, v) \} - \{ * ret=v * Val(x, v) \}} \text{ Frame}}{\{ ValRecell(x, v, o) \} - \{ ValRecell(x, v, o) * ret=v \}} \text{ Conseq}$$

This proof does *not* depend on either the internal representation of the `Cell` class or the body of the `get` method, only the static specification of the `Cell` class and the internal representation of the `Recell` class.

Second, we need to verify the proof obligation (Behavioural Subtyping), i.e. we must show that the `Recell`'s dynamic specification of the `get` method is a valid behavioural subtype of the `Cell`'s specification. This can be proved as follows.

$$\frac{\frac{\{ Val(this, v, o) \} - \{ Val(this, v, o) * ret=v \}}{\{ \exists o. Val(this, v, o) \} - \{ \exists o. Val(this, v, o) * ret=v \}} \text{ VarElim}}{\{ Val(this, v) \} - \{ Val(this, v) * ret=o \}} \text{ Conseq}$$

Note that this proof uses the arity manipulation described in §3.2.

Now, we turn our attention to the `set` method. The first proof obligation, (Body Verification), can be proved as follows.

```

{ ValRecell(this, v, ...) }
{ ValCell(this, v) * this.bak  $\mapsto$  ... }
tmp = this.Cell::get();
{ ValCell(this, v) * this.bak  $\mapsto$  ... * tmp = v }
this.bak = tmp;
{ ValCell(this, v) * this.bak  $\mapsto$  v }
this.Cell::set(x);
{ ValCell(this, x) * this.bak  $\mapsto$  v }
{ ValRecell(this, x, v) }

```

```

class TCell : Cell {
    int val2;

    define ValTCell(x, v) as ValCell(x, v) * x.val2 ↪ v

    TCell() dynamic {true} -{ ValTCell(this, _) } {}

    public override void set(int x)
        dynamic { Val(this, _) } -{ Val(this, x) }
        { this.val2=x; this.Cell::set(x); }

    public virtual void check()
        dynamic { Val(this, x) } -{ Val(this, x) }
        { int tmp = this.Cell::get(); if(this.val2 != tmp) crash(); }
}

```

Figure 4. The TCell code

The proof obligation (Behavioural Subtyping) is proved almost identically as for the get method.

$$\frac{\begin{array}{c} \{ Val(this, v, _) \} - \{ Val(this, x, v) \} \\ \{ \exists v. Val(this, v, _) \} - \{ \exists v. Val(this, x, v) \} \end{array}}{\{ Val(this, _) \} - \{ Val(this, x) \}} \text{ VarElim}$$

$$\frac{\{ \exists v. Val(this, v, _) \} - \{ \exists v. Val(this, x, v) \}}{\{ Val(this, _) \} - \{ Val(this, x) \}} \text{ Conseq}$$

The last proof obligation (Dynamic dispatch), can be shown simply and is omitted, as is the verification of the undo method.

Interestingly, this class can inherit all three versions of swap: we only need to provide proofs for swap1 and swap2. For swap1 we must show

$$\frac{\begin{array}{c} \{ Val_{Cell}(this, v₁) * Val(c, v₂) \} - \{ Val_{Cell}(this, v₂) * Val(c, v₁) \} \\ \{ Val_{Cell}(this, v₁) * Val(c, v₂) \} - \{ Val_{Cell}(this, v₂) * Val(c, v₁) \} \\ * this.bak ↪ _ * this : Recell \end{array}}{\begin{array}{c} \{ Val(this, v₁) * Val(c, v₂) \} - \{ Val(this, v₂) \} \\ * this : Recell \end{array}} - \{ \begin{array}{c} \{ Val(this, v₂) \} - \{ Val(c, v₁) \} \\ * Val(c, v₁) \end{array} \} \text{ Conseq}$$

For swap2 we must satisfy (1), which can be done trivially by defining ValLeft for Recell as $this.bak \rightarrow _$.

5.2 Restriction: TCell

Now we consider a derived class, TCell, that restricts the behaviour of its base class. The code is given in Figure 4. It defines a field val2 that is expected to contain the same value as would be returned by calling get. Every time the set method is called, both representations are updated, hence the check method should never be able to call crash.

For this class, we must prove that it is valid to inherit the get method, (Inheritance):

$$\frac{\begin{array}{c} \{ Val_{Cell}(x, v) \} - \{ Val_{Cell}(x, v) * ret=v \} \\ \{ Val_{Cell}(x, v) \} - \{ * ret=v * x.val2 \rightarrow v \} \end{array}}{\{ Val_{TCell}(x, v) \} - \{ Val_{TCell}(x, v) * ret=v \}} \text{ Frame}$$

$$\frac{\{ Val_{TCell}(x, v) \} - \{ Val_{TCell}(x, v) * ret=v \}}{\{ Val_{TCell}(x, v) \} - \{ Val_{TCell}(x, v) * ret=v \}} \text{ Conseq}$$

We will omit the proof obligations for the set method. The check method requires that we prove (Body Verification), which follows.

```

{ ValTCell(this, x) }
{ ValCell(this, x) * this.val2 ↪ x }
int tmp = this.Cell::get();
{ ValCell(this, x) * this.val2 ↪ x * tmp = x }
if(this.val2 != tmp) {
    { ValCell(this, x) * this.val2 ↪ x * tmp = x * tmp ≠ x }
    { false } crash(); { false }
    { ValCell(this, x) * this.val2 ↪ x * tmp = x }
}
{ ValCell(this, x) * this.val2 ↪ x * tmp = x }
{ ValTCell(this, x) }

```

```

class DCell : Cell {
    define ValDCell(x, v) as false
    define DValDCell(x, v) as ValCell(x, v)

    public inherit int get()
        dynamic { DVal(this, x) } -{ DVal(this, x) * ret = x }
        also { Val(this, x) } -{ Val(this, x) * ret = x }

    DCell() {} dynamic { true } -{ DValDCell(this, _) }

    public override void set(int x)
        dynamic { DVal(this, _) } -{ DVal(this, x * 2) }
        also { Val(this, _) } -{ Val(this, x) }
        { this.Cell::set(x * 2); }
}

```

Figure 5. The DCell class

Hence, the method can never call crash when its pre-condition is met.

Now, we consider inheriting the various swap methods. Firstly, swap can trivially be inherited. swap1 cannot be inherited:

$$\frac{\begin{array}{c} \{ Val_{Cell}(this, v₁) * Val(c, v₂) \} - \{ Val_{Cell}(this, v₂) * Val(c, v₁) \} \\ \{ Val_{Cell}(this, v₁) * Val(c, v₂) \} - \{ Val_{Cell}(this, v₂) * Val(c, v₁) \} \\ * this.val2 \rightarrow v₁ * this : TCell \end{array}}{\begin{array}{c} \{ Val_{Cell}(this, v₂) * Val(c, v₁) \} - \{ Val_{Cell}(this, v₁) * Val(c, v₂) \} \\ * this.val2 \rightarrow v₁ * this : TCell \end{array}} \text{ Conseq}$$

Our proof fails because $this.val2 \rightarrow v_1$, so we cannot establish the post-condition $Val(this, v_2) * Val(c, v_1)$ as this requires the field to have been updated to $this.val2 \rightarrow v_2$. Hence, TCell would have to override the swap1 method.

In addition, this class does not satisfy (1) required by swap2. The state separate from Val_{Cell} depends on the value v , but $ValLeft$ does not take this as an argument. If this constraint was imposed on the system, then the TCell class would not be allowed.

5.3 Reuse: DCell

In this example we present a subtype of Cell that is not a well-behaved subtype in the traditional sense of behavioural subtyping. We define the class DCell in Figure 5; it is essentially a Cell that doubles the value it is set to. This breaks the standard substitutivity property, and hence is not allowed in other verification methodologies. However, as we shall see, it can be verified in our proof system.

We define the predicate family for Val_{DCell} to be *false*. This prevents clients calling a DCell using the Cell's interface. This reflects the fact that the use of inheritance in DCell is for code reuse. To inherit the get method we must show the (Inheritance) and (Behavioural Subtyping) proof obligations. The former can be proved as follows.

$$\frac{\begin{array}{c} \{ Val_{Cell}(x, v) \} - \{ Val_{Cell}(x, v) * ret=v \} \\ \{ Val_{Cell}(x, v) * x:DCell \} - \{ Val_{Cell}(x, v) * ret=v * x:DCell \} \end{array}}{\{ DVal(x, v) * x:DCell \} - \{ DVal(x, v) * ret=v \}} \text{ Conseq}$$

The latter proof obligation follows directly from the definition of **also** (Definition 1).

Consider, the following client code of the DCell class.

```

public void crash()
dynamic { false } -{ false }
{ crash(); }

public void f(Cell c, DCell d)
dynamic { Val(c, _) * DVal(d, _) } -{ Val(c, 5) * DVal(d, 10) }
{ }

```

```

class SubRecell : Recell {
  Stack ints;

  define ValSubRecell(x, v1, v2, l) as
    x.ints  $\mapsto$  i * Stack(i, v1 :: l) * ((l = []  $\wedge$  v2 = v1)  $\vee$  l = v2 :: _)

  SubRecell() dynamic {true} -{ Val(this, _, _, _) }
  { ints = new Stack(); ints.push(0); }

  public override int get()
  dynamic { Val(this, v1, v2, l) } -{ Val(this, v1, v2, l) * v1 = ret }
  { return ints.readTop(); }

  public override void set(int x)
  dynamic { Val(this, v1, v2, l) } -{ Val(this, x, v1, v1 :: l) }
  { ints.push(x); }

  public override void undo()
  dynamic { Val(this, v1, v2, w2 :: l) } -{ Val(this, v2, _, l) * v2 = w2 }
  also { Val(this, v1, v2, []) } -{ Val(this, v2, _, []) }
  { if(ints.length() > 1) this.ints.pop(); }
}

```

Figure 6. Recell with unbounded backup

```

if(c.set(5).get() != 5) crash();
if(d.set(5).get() != 10) crash();
}

```

The specification for f amounts to showing that if the arguments c and d meet their specifications, then the method body never invokes the crash method. Ordinarily, this would be very hard to establish as DCell is not normally considered to be a behavioural subtype of Cell.

However, the power of the approach described in this paper is that it is possible (and quite simple) to show that the method f meets its specification. The verification proceeds directly from the dynamic specifications trivially. The first argument, c, can be any subtype of Cell that has the $Val(c, _)$ predicate. Hence, we cannot call this with the first argument of type DCell.

Surprisingly, the DCell class can validly inherit the swap1 method, and satisfy the constraint (1) for swap2. The constraint is trivially satisfied by defining $ValLeft$ for DCell as false. Similarly, the swap1 method can be inherited trivially, as the specifications pre-condition is false for this class, so the method will never be called.

5.4 Altering internal representation: SubRecell

Finally we consider an example where we completely alter how data is represented in the base class. In Figure 6, we present a derived class of Recell called SubRecell that has an unbounded undo capacity. The code uses a Stack to store the values, and does not update the redundant fields it inherits from Recell; the code is clearer and simpler by not using the parent fields. As all the methods are overridden this class can be rather straightforwardly verified. We must simply show that each method satisfies the rule for behavioural subtyping (Behavioural Subtyping), and the method bodies implement the specification (Body Verification).

The method swap can trivially be inherited as it does not depend on the representation. However, swap1 cannot be inherited as it assumes the original fields are used. The constraint, (1), required for swap2 cannot be satisfied by this class.

5.5 Interplay between static and dynamic calls

Interestingly, the set method of Recell becomes considerably harder to verify if the call to get is turned into a dynamic call, rather than a direct/super call, that is

```

void set(int x)
dynamic { Val(this, v, _) -{ Val(this, x, v) } }
static { Val(this, v, _) * Val_TORecell(this) } -{ Val(this, x, v) }
{ int tmp = this.get(); this.bak = tmp; this.Cell::set(x); }

```

The increased difficulty comes from considering all possible derived classes. If the derived class overrides the behaviour of get then this code could inflict untold damage! Hence, we must place a constraint on any method that inherits this code. This is done by adding Val_TO_{Recell} to the **static** pre-condition. We require that this predicate has the following properties

$$\begin{aligned} Val_TO_{Recell}(this) * Val(this, x, v) \\ \Rightarrow Val_{Recell}(this, x, v) * Val_FROM_{Recell}(this) \\ Val_{Recell}(this, x, v) * Val_FROM_{Recell}(this) \\ \Rightarrow Val_TO_{Recell}(this) * Val(this, x, v) \end{aligned}$$

If the derived class alters the representation too much, then it is not possible to find solutions to these equations. Finding these solutions can be simplified¹¹ to proving that the following is a tautology.

$$\forall xv. Val(this, x, v) -* \\ \left(\begin{array}{l} Val_{Recell}(this, x, v) * \\ \left(\forall x'v'. Val_{Recell}(this, x', v') -* Val(this, x', v') \right) \end{array} \right)$$

The outer use of $-*$ is the TO predicate, and the inner one is the FROM predicate. With this additional predicate we can perform the verification, (Body Verification), as follows:

```

{ Val(this, v, _) * Val_TORecell(this) }
  tmp = this.get();
{ Val(this, v, _) * Val_TORecell(this) * tmp = v }
{ Val_{Recell}(this, v, _) * Val\_FROM_{Recell}(this) * tmp=v }
{ ValCell(this, v) * this.bak  $\mapsto$  _ * Val\_FROM_{Recell}(this) * tmp=v }
  this.bak = tmp;
{ ValCell(this, v) * this.bak  $\mapsto$  v * Val\_FROM_{Recell}(this) }
  this.Cell::set(x);
{ ValCell(this, x) * this.bak  $\mapsto$  v * Val\_FROM_{Recell}(this) }
{ Val_{Recell}(this, x, v) * Val\_FROM_{Recell}(this) }
{ Val(this, x, v) }

```

6. Conclusions and related work

In this paper we have considered the problem of verifying object-oriented programs that use inheritance in a number of different ways. We have defined a proof system that allows a derived class to (1) simply extend a base class (Recell); (2) restrict its base class's behaviour (TCell); (3) alter its base class's behaviour in a way incompatible with the standard view of behavioural subtyping (DCell); and (4) replace the representation of its base class (SubRecell). Even in the presence of these drastic changes, we are still able to inherit code without needing to see the actual implementation. As far as we are aware, no other modular proof system can verify all of these examples.

Poetzsch-Heffter and Müller (1999) present a logic with rules with both virtual (dynamic) and implementation (static) specifications. However, they do not explore the inter-relationship between the virtual and implementation specifications, and they do not consider how this distinction enables the inheritance of methods without reverification. The interaction with inheritance is the key to the examples presented in this paper.

The Java Modelling Language, JML (Leavens et al. 2006), also has similar notions to static (known as code contracts) and dynamic

¹¹ Some might say simplifying to uses of $-*$ is not simplifying at all!

specifications (known as non-code behaviour specifications). However the treatment of these specifications is different. Code contracts can be used to verify method invocations where the exact method can be statically determined. JML also takes a different approach to verifying overridden methods. It requires a method in a derived class to be verified against not only its specification, but also against all the specifications in its base classes. This has the advantage of simplifying the framework (i.e. by eliminating proof-theoretic notions such as refinement), but our work is motivated by wishing to avoid such repeated verification of overridden methods.

Dhara and Leavens (1996) propose a relaxation of the JML approach by defining slightly more liberal restrictions on the pre- and post-conditions of a method in a derived class that ensure behavioural subtyping.

JML does not currently define restrictions on inheriting methods. Addressing this, Müller (2002) and later (Müller et al. 2006) restrict invariants in a derived class to only mention fields/properties introduced in that class, and the class must preserve the invariants of its base classes. This means that methods can be inherited, although not all code satisfies the restrictions. For example, this approach can only deal with the Cell, and Recell examples presented in this paper.

Ruby and Leavens (2000) allow the invariant of a derived class to depend on fields from its base class. They provide a series of conditions for when it is valid to inherit a method into a class. Their work can deal with Cell, Recell and TCell examples. It still requires a derived class to satisfy the invariant of its base classes, so it cannot allow representation changes that are required for SubRecell and DCell.

Spec[#]/Boogie (Barnett et al. 2004) also allows the invariant of a derived class to restrict the invariant of its base class. This means it can deal with the Cell, Recell and TCell examples. Spec[#]/Boogie uses a single, “polymorphic” specification for each method that is interpreted in one of two ways, one for static/direct dispatch, and one for the dynamic dispatch. They do not explicitly separate the specification as we do in this paper, but their approach is clearly closely related. A polymorphic specification is one containing a distinguished symbol, typically written “1”, that is replaced with the expression `type(this)` for dynamic dispatch, and with the expression `C` for the static dispatch, where `C` is the defining class.

This polymorphism can be justified quite succinctly using our notion of refinement, as follows (where we write `P[e]` to mean the formula `P` where all occurrences of the symbol `1` are replaced with the expression `e`):

$$\{P[C]\} \{-\{Q[C]\} \text{ this}^C \Rightarrow \{P[type(this)]\} \{-\{Q[type(this)]\}\}.$$

Currently, Spec[#]/Boogie cannot deal with the SubRecell and DCell classes as it enforces that a derived class preserves the invariant of its base class.

In future work, we intend to pursue a more thorough comparison of our approach and the Spec[#]/Boogie system. In joint work with others (Parkinson et al. 2007) we propose the classic “gang of four” design patterns as a benchmark for the verification of object-oriented code. These patterns often make use of complicated aggregate structures. Class invariant-based approaches, such as Spec[#], require significant extensions to handle these structures and their use (Leino and Schulte 2007). Early experiments suggest that our approach—using separation logic and abstract predicate families—requires no extensions to handle aggregate structures.

Note Independent to our work, Chin et al. (2008) suggest in these proceedings a similar approach to avoiding re-verification. Whilst working in the more traditional setting of class invariants, they propose a very similar use of static/dynamic method specifications in a separation logic. That two groups independently proposed the

distinction between static and dynamic specifications is perhaps encouraging as to the naturalness of the basic idea.

Acknowledgments

We should like to thank Wei-Ngan Chin and Sophia Drossopoulou for discussions on this work, and Gary Leavens, Rustan Leino, Peter Müller, Clyde Ruby and Wolfram Schulte for discussions about related work.

References

- M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
- M. Barnett, K. R. M. Leino, and W. Schulte. The Spec[#] programming system: An overview. In *Proceedings of CASSIS*, pages 49–69, 2005.
- B. Biering, L. Birkedal, and N. Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, 2007. To appear.
- G. M. Bierman, M. J. Parkinson, and A. M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2004.
- W.-N. Chin, C. David, H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *Proceedings of POPL*, 2008.
- W. R. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of POPL*, 1990.
- K. K. Dhara and G. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of ICSE*, 1996.
- C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of PLDI*, 1993.
- M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer’s reduction semantics for classes and mixins. Technical Report TR-97-293, Rice University, 1997. Corrected June, 1999.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
- S. Ishtiaq and P. W. O’Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, pages 14–26, 2001.
- N. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *Proceedings of FTfJP*, 2007.
- G. T. Leavens and D. A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report TR 06-36, Iowa State University, 2006.
- G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
- K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of OOPSLA*, pages 144–153, 1998.
- K. R. M. Leino and P. Müller. A verification methodology for model fields. In *Proceedings of ESOP*, 2006.
- K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *Proceedings of ESOP*, 2007.
- B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, 1994.
- P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.
- P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
- A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. In *Proceedings of ESOP*, 2007.

- P. W. O’Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, pages 1–19, 2001.
- M. Parkinson, G. Bierman, J. Noble, and W. Schulte. Contracts for patterns. Unpublished note, 2007.
- M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.
- M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.
- A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *Proceedings of ESOP*, volume 1576 of *LNCS*, 1999.
- J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74, 2002.
- C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. *SIGPLAN Not.*, 35(10):208–228, 2000.
- H. Yang. *Local reasoning for stateful programs*. PhD thesis, University of Illinois, July 2001.

A. Semantics of proof system

We give the semantics of our logic with respect to a state, σ , an interpretation of predicates, \mathcal{I} , and an interpretation of logical variables, \mathcal{L} . An interpretation of predicates maps predicate names to predicate definitions, where predicate definitions map a list of values to a set of states, that is:

$$\begin{aligned}\mathcal{I} : \text{Preds} &\rightarrow (\text{Vals}^* \rightarrow \mathcal{P}(\Sigma)) \\ \mathcal{L} : \text{Vars} &\rightarrow \text{Vals}\end{aligned}$$

We define predicates in the standard way for a predicate calculus:

$$\sigma, \mathcal{I}, \mathcal{L} \models \alpha(\bar{X}) \iff \sigma \in (\mathcal{I}(\alpha)(\mathcal{L}(\bar{X})))$$

Definition 6. $\mathcal{I} \models \Delta$ iff for all σ and \mathcal{L} then $\sigma, \mathcal{I}, \mathcal{L} \models \Delta$ holds

Given this definition, we can prove that any set of disjoint abstract predicate family definitions is satisfiable.

Lemma 7. For any set of disjoint definitions, $A_1 \dots A_n$, there exists an environment, \mathcal{I} , such that $\mathcal{I} \models [A_1] \wedge \dots \wedge [A_n]$, provided that if A_i defines α_C and A_j defines α_C , then $i = j$.

Next, we define the semantics of the judgements of the proof system. We use the standard semantics of a triple for separation logic, that is, if the pre-condition holds of the start state, then (1) the program will not fault (e.g. access unallocated memory); and (2) if the program terminates, then the final state will satisfy the post-condition.

Definition 8. $\mathcal{I} \models_n \{P\} \bar{s} \{Q\}$ iff $(S, H), \mathcal{I}, \mathcal{L} \models P$ then $\forall m \leq n$.

- $S, H, \bar{s} \xrightarrow{m} \text{fault does not hold; and}$
- if $S, H, \bar{s} \xrightarrow{m} S', H'$, skip then $(S', H'), \mathcal{I}, \mathcal{L} \models Q$.

Note that the step index n is used to deal with mutual recursion in method definitions.

We define $\mathcal{I} \models_n \Gamma$ to mean all the methods given in Γ meet their specifications for at least n steps.

Definition 9 (Semantics of method verification).

$$\begin{aligned}\mathcal{I}, \Gamma \models_{n+1} C.m(\bar{x}) : \{P\} - \{Q\} &\text{ iff} \\ \mathcal{I} \models_n \Gamma &\Rightarrow \mathcal{I} \models_{n+1} \{P \wedge \text{this} : C\} \text{ mbody}(C, m) \{Q\} \\ \mathcal{I}, \Gamma \models_{n+1} C::m(\bar{x}) : \{S\} - \{T\} &\text{ iff} \\ \mathcal{I} \models_n \Gamma &\Rightarrow \mathcal{I} \models_{n+1} \{S \wedge \text{this} \neq \text{null}\} \text{ mbody}(C, m) \{T\}\end{aligned}$$

$\mathcal{I} \models_0 \Gamma$ always holds.

$$\mathcal{I} \models_{n+1} \Gamma \text{ iff } \forall spec \in \Gamma. \mathcal{I}; \Gamma \models_{n+1} spec$$

We can now define the precise semantics of a judgement as follows.

Definition 10. $\Delta; \Gamma \models \{P\} \bar{s} \{Q\}$ iff for all \mathcal{I} and n , if $\mathcal{I} \models \Delta$ and $\mathcal{I} \models_n \Gamma$, then $\mathcal{I} \models_{n+1} \{P\} \bar{s} \{Q\}$.

That is, for all interpretations satisfying the predicate definitions Δ and assuming all the methods executed for at most n steps meet their specification given by Γ , then the statements \bar{s} meet their specification for at least $n + 1$ steps.

The judgements for statement and method verification are sound with respect to the semantics.

Lemma 11.

1. If $\Delta; \Gamma \vdash \{P\} \bar{s} \{Q\}$ then $\Delta; \Gamma \models \{P\} \bar{s} \{Q\}$.
2. If $\Delta; \Gamma \vdash M$ in E then $\forall \mathcal{I}. \text{if } \mathcal{I} \models \Delta \text{ then } \forall n \forall spec \in M. \mathcal{I}; \Gamma \models_n spec$

Our notion of refinement respects the weakening of assumptions and, hence, we can verify classes in a weaker context.

Lemma 12.

1. If $\Delta' \Rightarrow \Delta$ and $\Delta \vdash \{P_1\} - \{Q_1\} \Rightarrow \{P_2\} - \{Q_2\}$, then $\Delta' \vdash \{P_1\} - \{Q_1\} \Rightarrow \{P_2\} - \{Q_2\}$.
2. If $\Delta; \Gamma \vdash L$ and $\Delta' \Rightarrow \Delta$, then $\Delta'; \Gamma \vdash L$.

Finally, we state and outline the soundness proof for the program verification rule.

Theorem 13. If a program and main body \bar{s} can be proved using the program verification rule, then $\forall \mathcal{I}, n. \mathcal{I} \models_n \{\text{true}\} \bar{s} \{\text{true}\}$.

Proof. Using Lemma 12.2, we can simplify the rule to the following.

$$\frac{\Delta; \Gamma \vdash L_1 \quad \dots \quad \Delta; \Gamma \vdash L_n \quad \Delta; \Gamma \vdash \{\text{true}\} \bar{s} \{\text{true}\}}{\vdash L_1 \dots L_n \bar{s}}$$

where $\Gamma = \text{specs}(L_1 \dots L_n)$ and $\Delta = \text{apf}(L_1) \wedge \dots \wedge \text{apf}(L_n)$.

This rule assumes that Δ is satisfied, which we know by Lemma 7. The rest of the details are standard for the soundness of an object-oriented logic for partial correctness, for example see (Parkinson 2005). \square

An Object-Oriented Effects System^{*}

Aaron Greenhouse¹ and John Boyland²

¹ Carnegie Mellon University, Pittsburgh, PA 15213, aarong@cs.cmu.edu

² University of Wisconsin–Milwaukee, Milwaukee, WI 53201, boyland@cs.uwm.edu

Abstract. An effects systems describes how state may be accessed during the execution of some program component. This information is used to assist reasoning about a program, such as determining whether data dependencies may exist between two computations. We define an effects system for Java that preserves the abstraction facilities that make object-oriented programming languages attractive. Specifically, a subclass may extend abstract regions of mutable state inherited from the superclass. The effects system also permits an object’s state to contain the state of wholly-owned subsidiary objects. In this paper, we describe a set of annotations for declaring permitted effects in method headers, and show how the actual effects in a method body can be checked against the permitted effects.

1 Introduction

The *effects* of a computation include the reading and writing of mutable state. An effects system is an adjunct to a type system and includes the ability to infer the effects of a computation, to declare the permitted effects of a computation, and to check that the inferred effects are within the set of permitted effects.

The effects system we describe here is motivated by our desire to perform semantics-preserving program manipulations on Java source code. Many of the transformations we wish to implement change the order in which computations are executed. Assuming no other computations intervene and that each computation is single-entry single-exit, it is sufficient to require that the effects of the two computations do not *interfere*: one computation does not write state that is read or written by the other. Therefore our system only tracks reads and writes of mutable state, although other effects can be interesting (e.g., the allocation effect of FX [6]). Because Java supports separate compilation and dynamic loading of code, one of the goals of the project is to carry out transformations on

* This material is based upon work supported by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government. The work of John Boyland is supported by an equipment grant from Sun Microsystems, Inc.

incomplete programs. In earlier work [4], we proposed a system of program annotations including ones to declare the permitted effects of missing code. The effects system described in this paper uses and expands upon these annotations.

Declaring the (permitted) effects of a method necessarily constrains the implementation of the method and any method that overrides it, but we do not want to “break abstraction,” by revealing the implementations details of the program’s classes. One of the requirements for a useful object-oriented effects system is that it still provide the ability to hide names of private fields. It should use abstract names that map to multiple mutable locations. This paper introduces a concept of a named “region” in an object. The regions of an object provide a hierarchical covering of the notional state of the object. This idea is extended to objects which have the sole reference to other (“unique”) objects; the state of such owned objects is treated as extending the state of the owning object. As a result, a complex of numerous objects, such as a hash table, can be handled as a single notional object comprising a few regions of mutable state.

Our effects system uses declared annotations on fields and methods in Java. These annotations are non-executable, and can be ignored when implementing the program, although an optimizing compiler may find the information useful. The exact representation of the annotations is not important, but for the purposes of this paper, they are shown using an extension to Java syntax. The syntax of the effects annotations is given in Section 2.

A method body can be checked against its annotation using the annotations on the methods it calls. Similar to type-checking in the context of separate compilation, if all method bodies are checked at some point, the program as a whole obeys its annotations.

In the following section of the paper, we describe the basic concept behind the effects system: an abstract region of mutable state within an object. We also describe the technique for inferring the effects of a computation and checking inferred effects against declared effects. In the following section, we show how the regions of unique objects can be mapped into their owners. The precise description of the system is given in the appendix.

2 Regions of Objects

A *region* is an encapsulation of mutable state. The read and write effects of a method are reported with respect to the regions visible to the caller. In this section, we describe the general properties of a region, and how regions are specified by the programmer. Then we describe how methods are annotated, how the effects are computed, and how the annotation on a method can be checked for correctness.

2.1 Properties of Regions

The regions of a program are a hierarchy: at the root of the hierarchy there is a single region **All** that includes the complete mutable state of a program; at

the leaves of the hierarchy are all the (mutable) fields which again comprise the entire mutable state of the program.¹ Thus we see that each field is itself a region without child regions. The hierarchy permits us to describe the state accessed by a computation with varying precision: most precise is listing the individual fields accessed, least precise is `All`.

Java fields are declared in classes and are either `static` or not. In the latter case (instance fields), the declaration actually implies multiple fields, one per object of the class in which the field was declared. Our generalization of fields, regions, are similarly declared in classes in two ways: *static* declarations add one new region; *instance* declarations add a set of disjoint regions, one for each object of the class in which the region is declared. We call the non-field regions *abstract regions*.²

Each region is declared to be inside a parent region, thus creating the hierarchy. The parent region specified in a static region declaration must not refer to an instance region; such a static region would in effect have multiple parents. The root region `All` is declared static; one of its child regions is an instance region `Instance` declared in the root class `Object`.

Instance regions are inherited by subclasses, and a subclass may declare its own regions, possibly within an inherited (non-field) region. Being able to extend existing regions is critical to being able to specify the effects of a method and later being able to meaningfully override the method (see Example 2).

Arrays are treated as instances of a class `Array`, which has the instance region `[]` (pronounced “element”) as a subregion of `Instance`.³ When a subscripted element of an array is accessed, it is treated as an access of region `[]`.

The same accessibility modifiers that apply to field declarations (in Java, `public`, `protected`, default (package) access, or `private`) also apply to region declarations. The root region `All`, and its descendants `Instance` and `[]`, are `public`.

2.2 Specifying Regions

New syntax is used to declare abstract regions in class definitions. The syntax `region region` declares abstract instance regions. Adding the keyword `static` will declare an abstract static region. The parent of a region may be specified in the declaration by appending the annotation “*in parent*” to the field or abstract region declaration. Region declarations without an explicit parent are assigned to `All` or `Instance` respectively if they are static or not.

¹ External state (such as file state) is beyond the scope of this paper. Suffice it that special regions under `All` can be used to model this external state.

² We use the term “*abstract*” to emphasize the implementation hiding characteristics of these regions, not in the usual sense of Java “*abstract methods*” which impose implementation requirements on subclasses.

³ More precision could be obtained by distinguishing the different types of arrays, especially arrays of primitive types.

```

class Point {
    public region Position;
    private int x in Position;
    private int y in Position;
    public scale(int sc)
        reads nothing writes Position
    {
        x *= sc;
        y *= sc;
    }
}

class ColorPoint extends Point {
    public region Appearance;
    private int color in Appearance;
}

```

(a)

```

class Point3D extends Point {
    private int z in Position;
    public void scale(int sc)
        reads nothing writes Position
    {
        super.scale(sc);
        z *= sc;
    }
}

```

(b)

Fig. 1. The definitions of (a) classes `Point` and `ColorPoint`, and (b) class `Point3D`

The Java field selection syntax (`object.field` for instance fields and `Class.field` for static fields, as well as the provision for omitting `this` or the current class) is extended to all regions.

Example 1. Consider the class `Point` in Figure 1a. It declares the fields `x` and `y`, and the abstract region `Position` as their parent region. An instance `o` of class `Point` has four instance regions: `o.Instance`, which contains `o.Position`, which in turn contains `o.x` and `o.y`. The class `Point` has a method `scale`. This method is correctly annotated with `writes Position` (short for `writes this.Position`). The annotation `writes this.x`, `this.y` should not be used because the fields are private but the method is public. The annotation `writes Instance` would be legal, but less precise.

The class `ColorPoint` (also in Figure 1a) inherits from `Point`, declaring field `color` in a new abstract region `Appearance`, implicitly a subregion of `Instance`. If `ColorPoint` overrode `scale(int)`, the overriding method could not access `color` because `color` is not in `Position`, which is reasonable because `color` is not a geometric property. Given the less precise annotation on `scale(int)` mentioned above, the limitation would be lifted, because `color` is indeed in `Instance`.

We can create a second subclass of `Point`, `Point3D`, that adds the new field `z` to `Position` allowing it to be changed in the overriding of `scale(int)` as shown in Figure 1b.

2.3 Regions and Effects

As stated earlier, effects on objects are reported in terms of the regions that are affected. We distinguish between two kinds of effects: *read effects*, those that

```

annotation → reads targets writes targets
targets → nothing | targetSeq
targetSeq → target | target, targetSeq
target → var.region      region of a specific instance
          | any(class).region region of any instance of class class
          | class.region       (static) region of a class

```

var is restricted to being one of the method parameters or the receiver.

Fig. 2. The syntax of method annotations used in this paper.

may read the contents of a region; and *write effects*, those that *may* change or *read* the contents of a region. Making writing include reading is useful when overriding methods, as demonstrated in Examples 1 and 2. It also closely follows how an optional write would be modeled using static single-assignment form (SSA): as a merge of a read and a write.

Methods are labeled with their permitted effects using a notation similar to the Java **throws** clause. Figure 2 gives the syntax of method effect annotations. Again, analogous to **throws** clauses, the declared effects for a method describe everything that the method or *any of its overriding implementations* might affect. It is an error, therefore, for an overriding method to have more effects than declared for the method it overrides. The soundness of static analysis of the effects system would otherwise be compromised.

Example 2. This example shows the importance of abstract regions and the usefulness of having “write” include reading. Consider the class **Var** shown below, that encapsulates the reading and writing of a value. First we will show a set of annotations that do not permit a useful subclass, even assuming it were legal to expose names of private fields (regions) in annotations:

```

class Var {
    private int val;
    public void set( int x ) reads nothing writes this.val { val = x; }
    public int get() reads val writes nothing { return val; }
}

```

Suppose we now extended the class to remember its previous value. The annotations below are illegal because the declared effects of **UndoableVar.set()** are greater than the declared effects of **Var.set()**.

```

class UndoableVar extends Var {
    private int saved;
    public void set( int x ) reads nothing writes this.val, this.saved {
        saved = get(); super.set( x );
    }
    public void undo() reads this.saved writes this.val {
        super.set(saved);
    }
}

```

By instead placing the fields `val` and `saved` inside of a new abstract region `Value`, we are able to produce legal effect declarations:

```
class Var {
    public region Value;
    private int val in Value;
    public void set( int x ) reads nothing writes this.Value { val = x; }
    public int get() reads this.Value writes nothing { return val; }
}

class UndoableVar extends Var {
    private int saved in Value;
    public void set( int x ) reads nothing writes this.Value {
        saved = get(); super.set( x );
    }
    public void undo() reads nothing writes this.Value {
        super.set(saved);
    }
}
```

The implementation of `UndoableVar.set()` would not be legal if writing did not include the possibility of reading.

Computing Method Effects Effects are computed in a bottom-up traversal of the syntax tree. Every statement and expression has an associated set of effects. An effect is produced by any expression that reads or writes a mutable variable or mutable object field (in Java, immutability is indicated by the `final` modifier). The effects of expressions and statements include the union of the effects of all their sub-expressions. The computed effects use the most specific regions possible.

The analysis is intraprocedural; effects of method calls are obtained from the annotation on the called method, rather than from an analysis of the method's implementation. Because the method's declared effects are with respect to the formal parameters of the method, the effects of a particular method invocation are obtained by substituting the actual parameters for the formal parameters in the declared effects.

Checking Method Effects To check that a method has no more effects than permitted, each effect computed for the body of the method is checked against the effects permitted in the method's annotation. Any computed effect other than effects on local variables (which are irrelevant once the method returns) and effects on newly created objects must be *included* in at least one of the permitted effects in the annotation.⁴ The hierarchical nature of regions ensures that if an effect is included in a union of other effects then it must be included in one of the individual effects. Constructor annotations are not required to include write effects on the `Instance` region of the newly created object.

⁴ These exceptions can be seen as an instance of *effects masking* as defined for FX [6].

A read effect e_1 includes another read effect e_2 if e_1 's target includes e_2 's target. A write effect e_1 includes an effect e_2 if e_1 's target includes e_2 's target; e_2 may be a read or write effect. The target $var.region$ includes any target for the same instance and a region included in its region. The target $\text{any}(class).region$ includes any instance target referring to a region included in its region.⁵ The target $class.region$ includes any target using a region included by its region. Appendix A.3 defines effect inclusion precisely.

Example 3. Consider the body of `set` in class `UndoableVar`. In inferring the effects of the method body, we collect the following effects:

```
writes this.saved from this.saved = ...
reads this.Value from this.get()
writes this.Value from super.set(...)
reads x from x
```

The last effect can be ignored (it only involves local variables). The other three effects are all included in the effect `writes this.Value` in the annotation. This annotation is the same as that of the method it overrides

3 Unshared Fields

In this section we describe an additional feature of our effects system that permits the private state of an object to be further abstracted, hiding implementation details of an abstract data type from the user of the class. Ideally the user of the class `Vector` in Figure 3 should not care that it is implemented using an array, but any valid annotation of the `addElement` method based on what has been presented so far must reveal that its effects interfere with any access of an existing array. In this example, the array used to implement the vector is entirely encapsulated within the object, and so it is impossible that effects on this internal array could interfere with any other array.

Therefore, in addition to effect annotations, we now add the annotation `unshared` to the language as a modifier on field declarations. Unshared fields have the semantics of unique fields described in our earlier work [4] and formalized in a recently submitted paper [2]. Essentially an unshared field is one that has no aliases at all when it is read, that is, any object read from it is not accessible through any other variable (local, parameter or field).⁶ An analysis ensures this property partly by keeping track of (“dynamic”) aliases created by a read; the analysis rejects programs that may not uphold the property. We also use `unique` annotations on formal parameters and return values. The previously mentioned analysis ensures that the corresponding actual parameters are unaliased at the point of call and that a `unique` return value is unaliased when a method returns. For a more detailed comparison with the related terms used by other researchers, please see our related paper [2].

⁵ The class is used to resolve the region but not to define inclusion.

⁶ More precisely, the object *will not* be accessed through any other variable.

```

public class Vector {
    public region Elements, Size;
    private Object[] list in Elements;
    private int count in Size;

    public Vector() reads nothing writes nothing
    {
        this.list = new Object[10];
        this.count = 0;
    }

    public void addElement( Object obj )
        reads nothing writes this.Size, this.Elements, any(Array).[]
    {
        if( this.count == this.list.length ) {
            Object[] temp = new Object[this.list.length << 1];
            for( int i = 0; i < this.list.length; i++ )
                temp[i] = this.list[i];
            this.list = temp;
        }
        this.list[this.count++] = obj;
    }
    ...
}

```

Fig. 3. Dynamic array class implemented *without* unshared fields. Notice the exposure of `writes any(Array).[]` in method `addElement()`.

3.1 Using Unshared Fields

Let p refer to an object with an unshared field f that refers to a unique object. Because the object referenced by $p.f$ is *always* unaliased, it is reasonable to think of the contents of the referenced object as contents of the object p . Therefore, the instance regions of the object $p.f$ can be mapped into the regions of the object p . In this way, the existence of the object $p.f$ can be abstracted away. By default, the instance regions of the object referred to by an unshared field are mapped into the region that contains the unshared field (equivalent to mapping the region $p.f.Instance$ into the region $p.f$). This can be changed by the programmer, however. In the examples, we extend the syntax of unshared fields to map regions of the object referenced by the unshared field into regions of the object containing the field. The mapping is enclosed in braces $\{\}$. (See the declaration of the field `list` in Figure 4.)

The mapping of instance regions of the unshared field (more precisely, those regions of the object under `Instance`) must preserve the region hierarchy. Consider an unshared field f of object o that maps region $f.q$ into region $o.p$. If region $f.q'$ is a descendant of $f.q$, then it must be mapped into a descendant of $o.p$.

```

public class Vector {
    public region Elements, Size;
    private unshared Object[] list in Elements {Instance in Elements};
    private int count in Size;

    public Vector() reads nothing writes nothing
    {
        this.list = new Object[10];
        this.count = 0;
    }

    public void addElement( Object obj )
        reads nothing writes this.Size, this.Elements
    {
        if( this.count == this.list.length ) {
            Object[] temp = new Object[this.list.length << 1];
            for( int i = 0; i < this.list.length; i++ )
                temp[i] = this.list[i];
            this.list = temp;
        }
        this.list[this.count++] = obj;
    }
    ...
}

```

Fig. 4. Dynamic array class implemented using unshared fields. Differences from Figure 3 are underlined. Now `writes any(Array).[]` is absent from the annotation of method `addElement`.

Example 4. Consider a class `Vector`, shown in Figure 3 that implements a dynamic array. Because nothing is known about the possible aliasing of the field `list`, it must be assumed that the array is aliased, and that any use of an element of the array referenced by `list` interferes with any other array. There are two main reasons that this is undesirable. First, by requiring an effect annotation using the region `[]`, it is revealed that the dynamic array is implemented with an array. There is no reason that the user of the class needs to know this. Second, the required effect annotation prevents the following two statements from being swapped, even when `vector1` and `vector2` are not aliased (recall that we intend to use this analysis to support semantics-preserving program manipulations).

```

vector1.addElement( obj1 );
vector2.addElement( obj2 );

```

There is no reason for the user of the class to expect that two distinct dynamic arrays should interfere with each other. The problem is that the implementation of `Vector` does not indicate that two instantiations are necessarily independent, although a simple inspection of the code would reveal that they are. By making the `list` field `unshared`, as in Figure 4, this information is provided. Now the

`Instance` region (which includes the `[]` region) of `list` is mapped into into the region `Elements` of the `Vector` object, and it is no longer necessary to include the effect `writes Array []` in the annotation of `addElement`.

3.2 Checking Method Effects (Reprise)

In the previous section, we mentioned in passing that effects on local variables could be omitted when checking whether a method had no more effects than permitted. Effects on newly created objects can also be omitted, and similarly effects on unique references passed in as parameters can be omitted because by virtue of passing them, the caller retains no access (similar to “linear” objects) and thus any effects are irrelevant.

Unshared fields affect how we check inferred effects against permitted effects. Specifically, the declared mappings are used to convert any effects on regions of the unshared field into effects on the regions of the owner. This process is called “elaboration” and is detailed in Appendix A.4. The irrelevant effects are then masked from the elaborated effects and the remaining effects checked against those in the method’s annotation.

Example 5. Because we can use unshared fields to abstract away the implementation of a class, we can implement the same abstract data type in two different ways, but with both classes’ methods having the same annotations. Consider the implementation of a dynamic array shown in Figure 5. The field `head` in the class `LinkedVector` is unshared. This captures the notion that each dynamic array maintains a unique reference to its list of elements. This is necessary, but not sufficient, to make the effect annotations on `addElement` correct. It is not sufficient because it only indicates that the *first* element of the linked list is unshared; it says nothing about any of the other elements. Because it is the intention that each dynamic array maintains a distinct linked list, the `next` field of `Node` is declared to be `unshared` as well.

Let us verify the annotation of the method `addElement()` of class `Node`: `reads nothing writes this.Next`. The effect of the condition of the `if` statement is `reads this.next`, and the effect of the `then`-branch is `reads x, writes this.next`. The effect on the parameter `x` can be ignored outside the method, and the other effects are covered by the declared effect `writes this.Next`. The `else`-branch is more interesting, having a recursive call to `addElement()`. Looking up the declared effects of `addElement()`, we find `reads nothing writes this.Next`. The actual value of the receiver must now be substituted into the method effects. The receiver in this case is `this.next`, which is an unshared field whose `Next` region is mapped into region `this.Next`. The effect of the method call is thus `writes this.Next` (instead of a write to region `Next` of the object `this.next`), which is covered by the declared effects.

Checking that the declared effects of `addElement()` of `LinkedVector` are correct (and the same as the effects of `Vector.addElement()`) is now straightforward. The creation of a new node does not produce effects noticeable outside of the method. The conditional statement reads and writes `this.head`,

```

class Node {
    public region Obj, Next;
    public Object obj in Obj;
    public unshared Node next in Next
        {Instance in Next};

    public unique Node( Object o, Node n ) reads nothing writes nothing
    {
        this.obj = o;
        this.next = n;
    }

    public void addElement( unique Node x )
        reads nothing writes this.Next
    {
        if( this.next == null ) this.next = x;
        else this.next.addElement( x );
    }
}

class LinkedVector {
    public region Elements, Size;
    private unshared Node head in Elements
        { Instance in Elements };
    private int count in Size;

    public LinkedVector() reads nothing writes nothing
    {
        this.head = null;
        this.count = 0;
    }

    public void addElement( Object obj )
        reads nothing writes this.Elements, this.Size
    {
        Node tail = new Node( obj, null );
        if( this.head == null )
            this.head = tail;
        else
            this.head.addElement( tail );
        this.count += 1;
    }
    ...
}

```

Fig. 5. A dynamic array implemented using a linked list. Note that the annotation on `addElement(Object)` is the same as in Figure 4.

and calls `addElement()` on `this.head`. The first two effects are covered by the declared effect `writes this.Elements`, while the effect of the method call is `writes this.Elements` because the method call writes the `Next` region of the unshared field `this.head`, which maps `Next` to `this.Elements`. Finally, incrementing `this.count` has the effect `writes this.count`, which is covered by the declared effect `writes this.Size`.

Example 6. Figure 6 gives a simple implementation of a dictionary using association lists with annotations that abstract away the implementation details. It demonstrates the utility of being able to map the regions of an unshared field into multiple regions of the owner. The same set of annotations could be used for a hash table using the built-in hash code method `System.identityHashCode()`.

Assuming we had a variable `t` declared with type `Table`, the annotations permit the reordering of `t.containsKey()` and `t.pair()`, but do not permit reordering of `t.containsKey()` with `t.clear()`. The annotation on `clear()` is the most precise possible.

4 Using Effect Analysis

We will now briefly describe how the effects analysis can be used when applying semantics-preserving program manipulations. A precondition of many such manipulations (typically those that cause code motion) is that there not be any data dependencies between the parts of the program affected by the manipulation. In general, the two computations interfere if one mutates state also accessed by the other. Assuming the effects system is sound (see Section 6), then if the computations interfere, there is an effect from one computation that “conflicts” with an effect from the second. Two effects conflict if at least one is a write effect and they involve targets that may “overlap,” that is, may refer to the same mutable state at run time.

Two targets may overlap only if they refer to overlapping regions, and the hierarchical nature of regions ensures that regions overlap only if one is included in the other. Furthermore, an instance target (a use of an instance region of a particular object) can only overlap another instance target if the objects could be identical. The effect inference system computes effects (and targets) using the expressions in the computation in context. Thus equality cannot be compared devoid of context. This observation has led us to formalize the desired notion of equality in a new alias question *MayEqual*(e, e') [3]. Steensgaard’s “points-to” analysis [11] may be used as a conservative approximation. These considerations are further complicated by unshared fields. The details may be seen in the Appendix.

5 Related Work

Reynolds [10] showed how interference in Algol-like programs could be restricted using rules that prevent aliasing. His technique, while simple and general, requires access to the bodies of procedures being called in order to check whether

```

class Node {
    Object key, val;
    unshared Node next { Instance in Instance, key in key, val in val, next in next };

    unique Node(Object k, Object v, unique Node n) reads nothing writes nothing
    { key=k; val=v; next=n; }

    int count() reads next writes nothing
    { if (next == null) return 1; else return next.count()+1; }
    Object get(Object k) reads key, val, next writes nothing
    { if (k == key) return val; else if (next == null) return null;
      else return next.get(k); }

    .
    .

    boolean containsKey(Object k) reads key, next writes nothing
    { return k == key || next != null && next.containsKey(k); }
    void pair() reads key, next writes val
    { val = key; if (next != null) next.pair(); else ;}
}

public class Table {
    region Table;
    region Key in Table, Value in Table, Structure in Table;

    private unshared Node list in Structure
    { Instance in Instance, key in Key, val in Val, next in Structure };

    /** Return number of (key,value) pairs */
    public int count() reads Structure writes nothing
    { if (list == null) return 0;
      else return list.count(); }

    /** Remove all entries from table. */
    public void clear() reads nothing writes Structure { list = null; }

    public Object get(Object k) reads Table writes nothing
    { if (list == null) return null;
      else return list.get(k); }

    public void put(Object k, Object v) reads nothing writes Table { ... }
    public void remove(Object k) reads nothing writes Table { ... }
    public boolean contains(Object v) reads Val,Structure writes nothing { ... }

    public boolean containsKey(Object k) reads Key,Structure writes nothing
    { return list != null && list.containsKey(k); }

    /** Pair each existing key with itself. */
    public void pair() reads Key,Structure writes Value
    { if (list != null) list.pair(); else ;}
}

```

Fig. 6. Simple dictionary class with effects annotations.

they operate on overlapping global variables. The work includes a type system with mutable records, but the records cannot have recursive type (lists and trees are not possible).

Jackson’s Aspect system [7] uses a similar abstraction mechanism to our regions in order to specify the effects of routines on abstract data types. He uses specifications for each procedure in order to allow checking to proceed in a modular fashion. In his work, however, the specifications are *necessary* effects and are used to check for missing functionality.

Effects were first studied in the FX system [6], a higher-order functional language with reference cells. The burden of manual specification of effects is lifted through the use of effects inference as studied by Gifford, Jouvelot, and Talpin [8, 12]. The work was motivated by a desire to use stack allocation instead of heap allocation in mostly pure functional programs, and also to assist parallel code generation. The approach was demonstrated successfully for the former purpose in later work [1, 13]. These researchers also make use of a concept of disjoint “regions” of mutable state, but these regions are global, as opposed to within objects. In the original FX system, effects can be verified and exploited in separately developed program components.

Leino [9] defines “data groups,” which are sets of instance fields in a class. Each method declares which data groups it could modify. As with regions, a subclass may add new instance fields to existing data groups, but unlike our proposal, a field may be added to more than one data group. In their system, this ability is sound because the modifies clause is only used to see if a given method will modify a given field, not to see if effects of two methods could interfere.

Numerous researchers have examined the question of unique objects, as detailed in a paper of ours [2]. Most recently, Clarke, Potter and Noble have formalized a notion of “owner” that separates ownership from uniqueness [5]. Their notion of ownership appears useful for the kind of effects analysis we present here, although the precise mechanism remains further work.

6 Further Work

The work described in this paper is ongoing. We are continuing to extend our effects system to cover all of Java. We also intend to use the system to infer suggested annotations on methods. Ultimately, the effects system will be used to determine possible data dependencies between statements, which will require a form of alias analysis. This section sketches our ideas for further work.

An important aspect of a static effect system is for it to be *sound*, that is, it does not say two computations do not interfere through the access of some shared state when in fact they do. Proving soundness requires a run-time definition of interference which we must then show is conservatively approximated by our analysis. The run-time state of the program we are interested in includes fields of reachable objects as well as the state of local variables and temporaries in all activation frames. The effects of a computation can then be seen as reads and

writes on elements of the state. These effects can then be compared to that of the second computation. Interference occurs when an element of state is written by one computation and read or written by the other. We have begun formulating a proof of correctness using the concept of what run-time state is accounted for by a target (is “covered” by a target), proving that if a target includes another target than it covers everything covered by the other.

Java permits a limited form of multiple inheritance in the form of code-less “interfaces.” In order to permit useful annotations on method headers in interfaces, we must be able to add regions to interfaces and thus we must handle multiple inheritance of instance regions. (As with static fields, static regions are not inherited and thus do not complicate matters.) Multiple inheritance of instance regions is handled by permitting a class or interface inheriting regions from an interface to map them into other regions as long as the hierarchy is preserved. Any two regions, both of which are visible in a superclass or superinterface of the classes or interfaces performing the mapping, must have the same relation ($<$, $>$, or unrelated) in the latter class or interface. This restriction ensures that reasoning about regions remains sound. Conflicting relations in superclasses or superinterfaces may forbid certain inheritance combinations.

Java has *blank final* fields which are fields that are initialized in each constructor of the class that declared them but otherwise follow the restrictions on final fields—they may not be assigned. The Java language does not forbid methods from *reading* the value of a final field, even if the method is called in the constructor before the field is initialized. This situation is aggravated by classes which override methods called by the constructor in the superclass. In order to treat blank finals the same as finals (as immutable fields) and yet determine data dependencies correctly, extra annotations are needed on methods that may be called by a constructor. In particular, these methods must be annotated with the list of blank final fields that may be read. This annotation breaks abstraction to some degree but is needed only when checking the class itself and its subclasses.

Java supports multiple threads. So far we have ignored the possibility that state could be modified by a concurrent thread. As explained in a recent Java-World article [14], thread safety for the access of a field can be ensured in three situations:

1. the access is protected by a synchronization on the owning object;
2. the field is immutable (“final”);
3. the object has a “thread-safe” wrapper.

The first two cases can be easily checked through syntactic checks. The third condition is satisfied when the field is accessed through unshared or unique references.

As discussed in our earlier work [4], the task of adding annotations can be made less burdensome if it can be semi-automated. The same technique used to test whether a method stays within the permitted effects annotation can help to generate an annotation for an unannotated method. Essentially, the set of effects inferred from the method is culled by removing redundant effects, and

effects on regions inaccessible to the caller are promoted to effects on the nearest accessible parent region.

7 Conclusion

We have introduced an effects system for Java that permits an implementor to express the effects of a method in terms of abstract regions instead of in terms of mutable private fields. The effects system properly treats wholly-owned subsidiary objects as part of the owning object. We also introduce a way of checking the effects inferred from the body of a method against the effects permitted in a method’s annotation. As described in this paper, this effects system can be used to check consistency of annotations. It can also be used to determine whether there may be data dependencies between two computations involving separately compiled code.

Acknowledgments

We thank our colleagues at CMU (William L. Scherlis and Edwin C. Chan) and UWM (Adam Webber) for comments on drafts of the paper and numerous conversations on the topic. We also thank the anonymous reviewers.

References

1. Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN ’95 Conference on Programming Language Design and Implementation*, San Diego, California, USA, ACM SIGPLAN Notices, 30(6):174–185, June 1995.
2. John Boyland. Deferring destruction when reading unique variables. Submitted to IWAOS ’99, March 1999.
3. John Boyland and Aaron Greenhouse. MayEqual: A new alias question. Submitted to IWAOS ’99, March 1999.
4. Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE ’98)*, Kyoto, Japan, April 19–25, pages 167–176. IEEE Computer Society, Los Alamitos, California, 1998.
5. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA’98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, ACM SIGPLAN Notices, 33(10):48–64, October 1998.
6. D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.
7. Daniel Jackson. Aspect: Detecting bugs with abstract dependencies. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

8. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 303–310. ACM Press, New York, 1991.
9. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, ACM SIGPLAN Notices, 33(10), October 1998.
10. John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM Press, New York, January 1978.
11. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg, Florida, USA, January 21–24, pages 32–41. ACM Press, New York, 1996.
12. Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.
13. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Portland, Oregon, USA, January 17–21, pages 188–201. ACM Press, New York, 1994.
14. Bill Venners. Design for thread safety: Design tips on when and how to use synchronization, immutable objects, and thread-safe wrappers. *JavaWorld*, August 1998.

A Definitions

Throughout the appendix, class names/types and variable names are represented as elements of a set of identifiers **Ide**. We assume that facilities are available for determining the type/class of an expression/variable, and that the class hierarchy is available for determining if two classes are related. We use the phrase “class c is a descendant of class c' ” to mean that class c is equal to c' or c' ’s superclass is a descendant of c' .

$$\begin{array}{ll} \textit{var} \rightarrow \mathbf{Ide} & \textit{type} \rightarrow \mathbf{Ide} \\ \textit{name} \rightarrow \mathbf{Ide} & \textit{class} \rightarrow \mathbf{Ide} \end{array}$$

A.1 Syntax of Programming Language

Several language features used in the text have been omitted for simplicity; the remaining features can express the omitted features. The method receiver object is never implicit. It is illegal to write to a formal parameter. Parent regions must always be specified, and there are no default mappings for unshared fields. We require instance fields/regions to be explicitly declared as such using the keyword `instance`. A preprocessing pass can convert a program from the language presented in the body of the paper into the one presented here.

```

program → classDecl*
classDecl → class name extends name classBody
classBody → { classBodyDecl* }
classBodyDecl → region | field | method | constructor
region → kind region name in name;
field → kind type name in name;
| kind unshared type name in name {mappings};
mappings → mapping | mapping, mappings
mapping → name in name
kind → instance | static
method → type name ( params ) [annotation] block
constructor → name ( params ) [annotation] block
annotation → reads locations writes locations
locations → nothing | targetSeq
targetSeq → location | location, targetSeq
location → var . region | any(class) . region | class . region
params → ε | paramDecls
paramDecls → paramDecl | paramDecl, paramDecls
paramDecl → type name
block → { stmt* }
stmt → block | ; | return; | return expr ; | type name ; |
expr = expr; | stmtExpr; | constructObj(args); |
if( expr ) stmt else stmt | while( expr ) stmt
expr → name | rexpr | vexpr
rexpr → stmtExpr | allocExpr | constructObj | expr [expr] | expr.field
vexpr → expr ⊕ expr | ⊖expr | true | 1 | null | type
args → ε | argsSeq
argsSeq → expr | expr, argsSeq
stmtExpr → expr.method(args)
allocExpr → new type ([expr])+ | new class(args)
constructObj → super | this

```

A.2 Tracking locals

We need to track the mutable state references that could occur in local variables. The analysis here computes a relation between local variables and such reference expressions (*rexpr* in the preceding grammar) that each variable may be equal to. The relation also permits a local to be paired to the initial value of a formal parameter. It is not necessary that these expressions are available at the analysis point or even side-effect-free for this purpose. This analysis is very similar to standard def-use dataflow analyses. The relations will be drawn from a set **VB** for “variable bindings.”

$$\begin{aligned} \text{Bindings} &= \text{rexpr} + \text{Ide} \\ \mathbf{VB} &= 2^{\text{Ide} \times \text{Bindings}} \end{aligned}$$

$$\begin{aligned}
 B &: expr \rightarrow \mathbf{VB} \rightarrow \mathbf{Bindings} \\
 B \ name \ V &= \{b \mid (name, b) \in V\} \\
 B \ rexpr \ V &= \{rexpr\} \\
 B \ vexpr \ V &= \{\}
 \end{aligned}$$

$$V \setminus name = V - \{(name, b) \mid b \in \mathbf{Bindings}\}$$

For each statement s , we define two sets V_s^- and V_s^+ as the least fixed point solution to the following set of equations defined with regard to the syntax of each method or constructor:

$$\begin{aligned}
 &\dots \ name \ (\ params \) \ \dots \ block \\
 &\quad V_{block}^- = \{(var, var) \mid var \in params\} \\
 &block \rightarrow \{ \ stmt_1 \dots stmt_n \} \\
 &\quad V_{stmt_1}^-, \dots, V_{stmt_n}^-, V_{block}^+ = V_{block}^-, V_{stmt_1}^+, \dots, V_{stmt_n}^+ \\
 &stmt \rightarrow block \\
 &\quad V_{block}^-, V_{stmt}^+ = V_{stmt}^-, V_{block}^+ \\
 &stmt \rightarrow ; \\
 &\quad V_{stmt}^+ = V_{stmt}^- \\
 &stmt \rightarrow type \ name; \\
 &\quad V_{stmt}^+ = V_{stmt}^- \setminus name \\
 &stmt \rightarrow name = expr; \\
 &\quad V_{stmt}^+ = (V_{stmt}^- \setminus name) \cup \{(name, b) \mid b \in B \ expr \ V_{stmt}^-\} \\
 &stmt \rightarrow expr = expr; \quad (\text{Otherwise}) \\
 &\quad V_{stmt}^+ = V_{stmt}^- \\
 &stmt \rightarrow stmtExpr; \mid constructObj(args) \\
 &\quad V_{stmt}^+ = V_{stmt}^- \\
 &stmt \rightarrow if \ (\ expr \) \ stmt_1 \ else \ stmt_2 \\
 &\quad V_{stmt_1}^-, V_{stmt_2}^- = V_{stmt}^-, V_{stmt}^- \quad V_{stmt}^+ = V_{stmt_1}^+ \cup V_{stmt_2}^+ \\
 &stmt \rightarrow while \ (\ expr \) \ stmt' \\
 &\quad V_{stmt'}^-, V_{stmt}^+ = V_{stmt}^- \cup V_{stmt'}^+, V_{stmt}^- \cup V_{stmt'}^+
 \end{aligned}$$

When analyzing a target involving a local variable v , we will need the set of binding for that use. Let $B v$ be shorthand for $B v V_s^-$ where s is the immediately enclosing statement.

A.3 Formalization of Regions and Effects

A region is a triple $(class, name, tag)$, where $class$ is the class in which the region is declared, $name$ is the name of the region, and tag indicates whether the region is static or instance. We require the name of each region to be unique.

$$\begin{aligned}
 \mathbf{Regions}_I &= \mathbf{Ide} \times \mathbf{Ide} \times \{\text{instance}\} \quad (\text{Set of instance regions}) \\
 \mathbf{Regions}_S &= \mathbf{Ide} \times \mathbf{Ide} \times \{\text{static}\} \quad (\text{Set of static regions}) \\
 \mathbf{Regions} &= \mathbf{Regions}_I \cup \mathbf{Regions}_S = \mathbf{Ide} \times \mathbf{Ide} \times \{\text{static, instance}\}
 \end{aligned}$$

An effect is a read of or write to one of four kinds of targets, which represent different granularities of state.

$$\text{effect} \rightarrow \text{read}(\text{target}) \mid \text{write}(\text{target})$$

$$\text{target} \rightarrow \text{local Ide} \mid \text{instance } \text{expr} \times \mathbf{Regions}_I \mid \text{anyInstance } \mathbf{Regions}_I \mid \text{static } \mathbf{Regions}_S$$

The region hierarchy is a triple $(\text{regions}, \text{parent}_R, \text{unshared})$, where regions is the set of regions in the program, parent_R is the parent ordering of the regions; $r \text{parent}_R s \Leftrightarrow s$ is the super region of r ; and unshared is the mapping of regions of unshared fields to regions of the class. The mapping is a set of 4-tuples $(\text{class}, \text{field}, \text{region}_u, \text{region}_c)$ with the interpretation that field is an unshared field of instances of class class , and region region_u of the object referenced the field is mapped into region region_c of the object containing field .

$$\mathbf{Map} = \mathbf{Ide} \times \mathbf{Ide} \times \mathbf{Regions} \times \mathbf{Regions}$$

$$\mathbf{Hier} = 2^{\mathbf{Regions}} \times (\mathbf{Regions} \rightarrow \mathbf{Regions}) \times 2^{\mathbf{Map}}$$

$$\mathbf{All} = (\mathbf{Object}, \mathbf{All}, \mathbf{static})$$

$$\mathbf{Instance} = (\mathbf{Object}, \mathbf{Instance}, \mathbf{instance})$$

$$\mathbf{Element} = (\mathbf{Array}, [], \mathbf{instance})$$

$$\mathbf{H}_0 = (\{\mathbf{All}, \mathbf{Instance}, \mathbf{Element}\}, \{(\mathbf{Instance}, \mathbf{All}), (\mathbf{Element}, \mathbf{Instance})\}, \emptyset)$$

$$\mathbf{lookup} : 2^{\mathbf{Regions}} \rightarrow \mathbf{Ide} \rightarrow \mathbf{Ide} \rightarrow \mathbf{Regions}$$

$$\mathbf{lookup} r c f = (c', f, t) \in r \text{ s.t. class } c \text{ is a descendant of class } c'$$

The region hierarchy $(\text{rgns}, \text{parent}_R, m)$ is built from \mathbf{H}_0 , the initial hierarchy, by analyzing the field and region declarations in the program's class definitions. It must have the following properties: no region may have more than one parent; the region **Instance** of an unshared field must be mapped into a region of the owning object; the unshared mappings must respect the tree structure; only instance regions of unshared fields may be mapped; and regions of a static unshared field may only be mapped into static regions.

$$\begin{aligned} & \forall (c, r, p) \in \text{rgns}. \forall (c, r, p') \in \text{rgns}. p = p' \\ & (\exists (c, f, r_u, r_c) \in m) \Rightarrow (\exists (c, f, \mathbf{Instance}, q) \in m) \\ & \forall (c, f, r_u, r_c) \in m. \left(\begin{array}{l} (\forall (c, f, r'_u, r'_c) \in m. r'_u \leq_R r_u \Rightarrow r'_c \leq_R r_c) \\ \wedge (r_u \leq_R \mathbf{Instance}) \wedge (f \in \mathbf{Regions}_S \Rightarrow r_c \in \mathbf{Regions}_S) \end{array} \right) \end{aligned}$$

Inclusion Given the region hierarchy $(\text{rgns}, \text{parent}_R, m)$, we create the reflexive transitive closure of parent_R to define the inclusion relation over regions.

$$\leq_R \subseteq \mathbf{Regions} \times \mathbf{Regions}$$

$$\leq_R = (\text{parent}_R \cup \{(r, r) \mid r \in \text{rgns}\})^*$$

The inclusion relation over targets, \leq_T , is defined to be the reflexive transitive closure of a relation, parent_T .

$$\begin{aligned}
\text{unshared } e &\Leftrightarrow e = e' \cdot f \wedge \exists(c, f, r_u, r_c) \in m \\
\text{shared } e &\Leftrightarrow \neg \text{unshared } e \\
\text{parent}_T &\subseteq \text{target} \times \text{target} \\
\text{local } v \text{ parent}_T \text{ local } w &\Leftrightarrow v = w \\
\text{instance}(e, r) \text{ parent}_T \text{ instance}(e, r') &\Leftrightarrow r \text{ parent}_R r' \\
\text{instance}(e.f, r) \text{ parent}_T \text{ instance}(e, r') &\Leftrightarrow \text{lookup rgns}(\text{typeof } e) f \in \text{Regions}_I \\
&\quad \wedge \exists(c, f, r, r') \in m \\
\text{instance}(e, r) \text{ parent}_T \text{ anyInstance } r &\Leftrightarrow \text{shared } e \\
\text{instance}(e.f, r) \text{ parent}_T \text{ static } s &\Leftrightarrow \text{lookup rgns}(\text{typeof } e) f \in \text{Regions}_S \\
&\quad \wedge \exists(c, f, r, s) \in m \\
\text{anyInstance } r \text{ parent}_T \text{ anyInstance } r' &\Leftrightarrow r \text{ parent}_R r' \\
\text{anyInstance } r \text{ parent}_T \text{ static } s &\Leftrightarrow r \text{ parent}_R s \\
\text{static } s \text{ parent}_T \text{ static } s' &\Leftrightarrow s \text{ parent}_R s' \\
\\
\leq_T &\subseteq \text{target} \times \text{target} \\
\leq_T = &(\text{parent}_T \cup \{(t, t) \mid t \in \text{target}\})^*
\end{aligned}$$

Finally, effect inclusion is defined using \leq_T in the obvious manner.

$$\begin{aligned}
\leq_E &\subseteq \text{effect} \times \text{effect} \\
\text{read}(t) \leq_E \text{read}(t') &\Leftrightarrow t \leq_T t' \\
\text{write}(t) \leq_E \text{write}(t') &\Leftrightarrow t \leq_T t' \\
\text{read}(t) \leq_E \text{write}(t') &\Leftrightarrow t \leq_T t'
\end{aligned}$$

Method Annotations The method annotations are used to construct a method dictionary: a set of tuples $(\text{class}, \text{method}, \text{params}, \text{effects})$, where $\text{class}.\text{method}$ is the method (or constructor) name, params is a vector of type–identifier pairs that captures the method’s signature and formal parameter names, and effects is the set of effects that the method produces. The method dictionary is produced by analyzing the annotations of the methods in each defined class. For a program p , with region hierarchy (r, parent_R, m) , the set of method dictionary is $\mathcal{A}_P[p] \cap r \emptyset$.

$$\begin{aligned}
\text{Method} &= \text{Ide} \times \text{Ide} \times (\text{Ide} \times \text{Ide})^* \times 2^{\text{effect}} \\
\mathcal{A}_P &: \text{program} \rightarrow 2^{\text{Regions}} \rightarrow 2^{\text{Method}} \rightarrow 2^{\text{Method}} \\
\mathcal{A}_C &: \text{classDecl} \rightarrow 2^{\text{Regions}} \rightarrow 2^{\text{Method}} \rightarrow 2^{\text{Method}} \\
\mathcal{A}_B &: \text{classBody} \rightarrow 2^{\text{Regions}} \rightarrow \text{Ide} \rightarrow 2^{\text{Method}} \rightarrow 2^{\text{Method}} \\
\mathcal{A}_D &: \text{classBodyDecl} \rightarrow 2^{\text{Regions}} \rightarrow \text{Ide} \rightarrow 2^{\text{Method}} \rightarrow 2^{\text{Method}} \\
\mathcal{A}_A &: \text{annotation} \rightarrow 2^{\text{Regions}} \rightarrow 2^{\text{effect}} \\
\mathcal{A}_T &: \text{locations} \rightarrow 2^{\text{Regions}} \rightarrow 2^{\text{target}} \\
\mathcal{A}_F &: \text{params} \rightarrow (\text{Ide} \times \text{Ide})^*
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}_P[\text{classDecl}^*]r a &= (\mathcal{A}_C[\text{classDecl}_n]r) \cdots (\mathcal{A}_C[\text{classDecl}_1]r a) \\
\mathcal{A}_C[\text{class } c \text{ extends } p \text{ classBody}]r a &= \mathcal{A}_B[\text{classBody}]r c a \\
\mathcal{A}_B[\{\text{classBodyDecl}^*\}]r c a &= (\mathcal{A}_D[\text{classBodyDecl}_n]r c) \\
&\quad \cdots (\mathcal{A}_D[\text{classBodyDecl}_1]r c) a \\
\mathcal{A}_D[\text{region}]r c a &= a \\
\mathcal{A}_D[\text{field}]r c a &= a \\
\mathcal{A}_D[\text{type name (params)} \text{ anno } b]r c a &= a \cup \{(c, \text{name}, \mathcal{A}_F[\text{params}], \mathcal{A}_A[\text{anno}]r)\} \\
\mathcal{A}_D[c \text{ (params)} \text{ anno } b]r c a &= a \cup \{(c, c, \mathcal{A}_F[\text{params}], \mathcal{A}_A[\text{anno}]r)\} \\
\mathcal{A}_A[\epsilon]r &= \{\text{writes(All)}\} \\
\mathcal{A}_A[\text{reads } locs_R \text{ writes } locs_W]r &= \bigcup_{t \in \mathcal{A}_T[\text{locs}_R]r} \text{read}(t) \cup \bigcup_{t \in \mathcal{A}_T[\text{locs}_W]r} \text{write}(t) \\
\mathcal{A}_T[\text{nothing}]r &= \emptyset \\
\mathcal{A}_T[\text{location, targetSeq}]r &= \mathcal{A}_T[\text{location}]r \cup \mathcal{A}_T[\text{targetSeq}]r \\
\mathcal{A}_T[\text{var.region}]r &= \{\text{instance (var, lookup } r \text{ (typeof var) region)}\} \\
\mathcal{A}_T[\text{class.region}]r &= \{\text{static (lookup } r \text{ class region)}\} \\
\mathcal{A}_T[\text{any(class).region}]r &= \{\text{anyInstance (lookup } r \text{ class region)}\} \\
\mathcal{A}_F[\epsilon] &= () \\
\mathcal{A}_F[\text{type name}] &= (\text{type, name}) \\
\mathcal{A}_F[\text{type name, params}] &= ((\text{type, name}) : \mathcal{A}_P[\text{params}]) \\
\\
\text{typeof} &: \text{expr} \rightarrow \text{Ide} \\
\text{typeof } e &= \text{static class of expression } e
\end{aligned}$$

A.4 Computing Effects

The effects of an expression e in program p , with region hierarchy (r, parent_R, m) and effects dictionary d , are $\mathcal{E}[e] (r, \text{parent}_R, m) d$.

$$\mathcal{E}[-] : \text{Hier} \rightarrow 2^{\text{Method}} \rightarrow 2^{\text{effect}}$$

Features that do not have effects

$$\begin{aligned}
\mathcal{E}[\text{;}]h d &= \emptyset & \mathcal{E}[\text{type}]h d &= \emptyset & \mathcal{E}[\text{true}]h d &= \emptyset \\
\mathcal{E}[\text{return;}]h d &= \emptyset & \mathcal{E}[\text{1}]h d &= \emptyset & \mathcal{E}[\text{null}]h d &= \emptyset
\end{aligned}$$

Features that do not have direct effects

$$\begin{aligned}
\mathcal{E}[\{\text{stmt}^*\}]h d &= \bigcup \mathcal{E}[\text{stmt}_i]h d \\
\mathcal{E}[\text{return } expr;]h d &= \mathcal{E}[expr]h d \\
\mathcal{E}[expr_1 = expr_2;]h d &= \mathcal{E}_{\text{LVal}}[expr_1]h d \cup \mathcal{E}[expr_2]h d \\
\mathcal{E}[\text{if(} expr \text{) stmt}_1 \text{ else stmt}_2]h d &= \mathcal{E}[expr]h d \cup \mathcal{E}[\text{stmt}_1]h d \cup \mathcal{E}[\text{stmt}_2]h d \\
\mathcal{E}[\text{while(} expr \text{) stmt}]h d &= \mathcal{E}[expr]h d \cup \mathcal{E}[\text{stmt}]h d \\
\mathcal{E}[expr_1 \oplus expr_2]h d &= \mathcal{E}[expr_1]h d \cup \mathcal{E}[expr_2]h d \\
\mathcal{E}[\ominus expr]h d &= \mathcal{E}[expr]h d \\
\mathcal{E}[expr, \text{argsSeq}]h d &= \mathcal{E}[expr]h d \cup \mathcal{E}[\text{argsSeq}]h d \\
\mathcal{E}[\text{new type ([expr])}^+]h d &= \bigcup \mathcal{E}[\text{expr}_i]h d
\end{aligned}$$

Features that have direct effects Reading or writing to a local variable produces an effect on a local target. Array subscripting produces an effect on the array's region `[]`. Accessing a field produces an effect on an instance or static target, depending on whether the field is an instance or static field, respectively.

$$\begin{aligned}\mathcal{E}[\![\text{var}]\!] h d &= \{\text{read(local var)}\} \\ \mathcal{E}[\![\text{constructObj}]\!] h d &= \{\text{read(local this)}\} \\ \mathcal{E}[\![\text{expr}_1[\text{expr}_2]]\!] h d &= \mathcal{E}[\![\text{expr}_1]\!] h d \cup \mathcal{E}[\![\text{expr}_2]\!] h d \\ &\quad \cup \{\text{read(instance(expr₁, Element))}\} \\ \mathcal{E}[\![\text{expr.field}]\!] (r, \text{parent}_R, m) d &= \mathcal{E}[\![\text{expr}]\!](r, \text{parent}_R, m) d \cup \{\text{read}(t)\}\end{aligned}$$

where $\text{rgn} = \text{lookup } r \text{ (typeof expr) field}$

$$t = \begin{cases} \text{instance(expr, rgn)} & \text{if } \text{rgn} \in \mathbf{Regions}_I \\ \text{static rgn} & \text{if } \text{rgn} \in \mathbf{Regions}_S \end{cases}$$

$$\begin{aligned}\mathcal{E}_{\text{LVal}}[\![\text{var}]\!] h d &= \{\text{write(local var)}\} \\ \mathcal{E}_{\text{LVal}}[\![\text{expr}_1[\text{expr}_2]]\!] h d &= \mathcal{E}[\![\text{expr}_1]\!] h d \cup \mathcal{E}[\![\text{expr}_2]\!] h d \\ &\quad \cup \{\text{write(instance(expr₁, Element))}\} \\ \mathcal{E}_{\text{LVal}}[\![\text{expr.field}]\!] (r, \text{parent}_R, m) d &= \mathcal{E}[\![\text{expr}]\!](r, \text{parent}_R, m) d \cup \{\text{write}(t)\}\end{aligned}$$

where $\text{rgn} = \text{lookup } r \text{ (typeof expr) field}$

$$t = \begin{cases} \text{instance(expr, rgn)} & \text{if } \text{rgn} \in \mathbf{Regions}_I \\ \text{static rgn} & \text{if } \text{rgn} \in \mathbf{Regions}_S \end{cases}$$

Method and constructor calls In addition to any effects produced by evaluating the actual parameters and the receiver, method calls also have the effects produced by the method. These are found by first looking up the method being called in the method dictionary (using `find`, which uses the language specific static method resolution semantics), and then substituting the expression of the actual parameter for the formal in any effects with instance targets. The helper function `process` creates a vector of type-expression pairs out of the argument list; the types are used in `find`, while the expressions are used in `pair`. The function `pair` is used to create a set of formal–actual parameter pairs. The actual substitutions are made by the function `replace`.

$$\begin{aligned}\mathcal{E}[\![\text{expr.method(args)}]\!] h d &= \mathcal{E}[\![\text{expr}]\!] h d \cup \mathcal{E}[\![\text{args}]\!] h d \\ &\quad \cup \text{replace } e \text{ (pair } p \text{ l)} \cup \{(\text{this}, \text{expr})\} \\ \text{where } l &= \text{process args} \\ (c, \text{method}, p, e) &= \text{find } d \text{ (typeof expr) method } l\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![\text{new class(args)}]\!] h d &= \mathcal{E}[\![\text{args}]\!] h d \cup \text{replace } e \text{ (pair } p \text{ l)} \\ \text{where } l &= \text{process args} \\ (class, \text{class}, p, e) &= \text{find } d \text{ class class } l\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![\text{constructObj(args)}]\!] h d &= \mathcal{E}[\![\text{args}]\!] h d \cup \text{replace } e \text{ (pair } p \text{ l)} \\ \text{where } l &= \text{process args} \\ (c, \text{method}, p, e) &= \text{find } d \text{ (typeof constructObj) (typeof constructObj) } l\end{aligned}$$

```

find :  $2^{\text{Method}} \rightarrow \mathbf{Ide} \rightarrow \mathbf{Ide} \rightarrow (\mathbf{Ide} \times \mathbf{Ide})^* \rightarrow \text{Method}$ 
find  $d\ c\ m\ l = (c', m, p, e) \in d$  subject to language's
      static method resolution mechanism

process :  $\text{args} \rightarrow (\mathbf{Ide} \times \text{expr})^*$ 
process  $\epsilon = ()$ 
process  $\text{expr} = (\text{typeof } \text{expr}, \text{expr})$ 
process  $\text{expr}, \text{argSeq} = ((\text{process } \text{expr}) : (\text{process } \text{argSeq}))$ 

pair :  $(\mathbf{Ide} \times \mathbf{Ide})^* \rightarrow (\mathbf{Ide} \times \text{expr})^* \rightarrow 2^{\mathbf{Ide} \times \text{expr}}$ 
pair  $()\ () = \emptyset$ 
pair  $((t, n) : p)\ ((t', e) : l) = \{(n, e)\} \cup \text{pair } p\ l$ 

replace :  $2^{\text{effect}} \rightarrow 2^{\mathbf{Ide} \times \text{expr}} \rightarrow 2^{\text{effect}}$ 
replace  $e\ p = R' \cup W' \cup (e - (R \cup W))$ 
  where  $R = \{\text{read}((f, r)) \in e \mid (f, a) \in p\}$ 
   $W = \{\text{write}((f, r)) \in e \mid (f, a) \in p\}$ 
   $R' = \{\text{read}((a, r)) \mid \text{read}((f, r)) \in R \wedge (f, a) \in p\}$ 
   $W' = \{\text{write}((a, r)) \mid \text{write}((f, r)) \in W \wedge (f, a) \in p\}$ 

```

Elaborating Effects and Masking Sometimes we need to expand a set of effects to take into account the bindings of local variables and the mapping of unshared fields. In particular we need this *elaboration* when we check a method annotation.

$$\begin{aligned} \text{elaborate}(E) = & \left\{ \text{read}(t) \mid t \in \text{elaborate}(\{t'\}), \text{read}(t') \in E \right\} \\ & \cup \left\{ \text{write}(t) \mid t \in \text{elaborate}(\{t'\}), \text{write}(t') \in E \right\} \end{aligned}$$

$$\begin{aligned} \text{elaborate}(T) = & \text{smallest set } T' \supseteq T \text{ such that} \\ & (\text{instance}(v, r) \in T', e \in B v) \Rightarrow \text{instance}(e, r) \in T' \\ & \text{instance}(e.f, r) \in T' \Rightarrow \text{instance}(e, r') \in T' \\ & \text{where } r' = \min_{\leq_R} \{r_c \mid (c, f, r_u, r_c) \in m, r \leq_R r_u\} \end{aligned}$$

Effect masking is accomplished by dropping out effects on local variables, regions of local variables (handled in elaboration), regions in newly allocated objects, and regions of unshared fields (which are redundant after elaboration):

$$\begin{aligned} \text{mask}(E) = & \left\{ \text{read}(t) \mid t \in \text{mask}(\{t'\}), \text{read}(t') \in E \right\} \\ & \cup \left\{ \text{write}(t) \mid t \in \text{mask}(\{t'\}), \text{write}(t') \in E \right\} \end{aligned}$$

$$\text{mask}(T) = \{t \in T \mid t \text{ not of the forms } \text{local } v \text{ (where } v \text{ is any local or parameter),} \\ \text{instance } (v, r) \text{ (where } v \text{ is a local variable, but not a parameter)} \\ \text{instance } (\text{allocExpr}, r), \text{ or instance } (e, r) \text{ (where unshared } e)\}$$

A.5 Checking Annotated Effects

The annotated effects of a method, E_A , need to be checked against the computed effects of the method, E_C . If E_A does not account for every possible effect of the method, the annotation is invalid. Formally, the annotated effects of a method are valid if and only if $\forall e \in \text{mask}(\text{elaborate}(E_C)). \exists e' \in E_A. e \leq_E e'$

A.6 Conflicts and Interference

Two effects *conflict* if and only if $\text{read}(t_1)$ is included (by \leq_E) in one of the effects, $\text{write}(t_2)$ is included in the other effect, and $t_1 \text{overlap}_T t_2$. This can be extended to sets of effects: E_1 and E_2 conflict if and only if $\exists e_1 \in \text{elaborate}(E_1). \exists e_2 \in \text{elaborate}(E_2). e_1$ and e_2 conflict. Finally, two computations *interfere* if and only if they have conflicting sets of effects. Target overlap, overlap_T , is the symmetric closure of overlap_0 .

$$\begin{aligned} \text{overlap}_R &\subseteq \text{Regions} \times \text{Regions} \\ r \text{overlap}_R r' &\Leftrightarrow r \leq_R r' \vee r' \leq_R r \\ \text{overlap}_T &\subseteq \text{target} \times \text{target} \\ \text{overlap}_T &= \text{overlap}_0 \cup \{(t', t) \mid (t, t') \in \text{overlap}_0\} \end{aligned}$$

$$\begin{aligned} \text{overlap}_0 &\subseteq \text{target} \times \text{target} \\ \text{local } v \text{overlap}_0 \text{local } w &\Leftarrow v = w \\ \text{instance } (e, r) \text{overlap}_0 \text{instance } (e', r') &\Leftarrow \text{MayEqual}(e, e') \wedge r \text{overlap}_R r' \\ \text{instance } (e, r) \text{overlap}_0 \text{anyInstance } r' &\Leftarrow \text{shared } e \wedge r \text{overlap}_R r' \\ \text{instance } (e, r) \text{overlap}_0 \text{static } s &\Leftarrow \text{shared } e \wedge r \text{overlap}_R s \\ \text{anyInstance } r \text{overlap}_0 \text{anyInstance } r' &\Leftarrow r \text{overlap}_R r' \\ \text{anyInstance } r \text{overlap}_0 \text{static } s &\Leftarrow r \text{overlap}_R s \\ \text{static } s \text{overlap}_0 \text{static } s' &\Leftarrow s \text{overlap}_R s' \end{aligned}$$

A Type System for Borrowing Permissions

Karl Naden Robert Bocchino

Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA, USA
{kbn,rbochin,jonathan.aldrich}@cs.cmu.edu

Kevin Bierhoff

Two Sigma Investments, New York, NY, USA

kevin.bierhoff@cs.cmu.edu

Abstract

In object-oriented programming, unique permissions to object references are useful for checking correctness properties such as consistency of typestate and noninterference of concurrency. To be usable, unique permissions must be *borrowed* — for example, one must be able to read a unique reference out of a field, use it for something, and put it back. While one can null out the field and later reassign it, this paradigm is ungainly and requires unnecessary writes, potentially hurting cache performance. Therefore, in practice borrowing must occur in the type system, without requiring memory updates. Previous systems support borrowing with external alias analysis and/or explicit programmer management of *fractional permissions*. While these approaches are powerful, they are also awkward and difficult for programmers to understand. We present an integrated language and type system with unique, immutable, and shared permissions, together with new *local permissions* that say that a reference may not be stored to the heap. Our system also includes *change permissions* such as `unique>>unique` and `unique>>none` that describe how permissions flow in and out of method formal parameters. Together, these features support common patterns of borrowing, including borrowing multiple local permissions from a unique reference and recovering the unique reference when the local permissions go out of scope, without any explicit management of fractions in the source language. All accounting of fractional permissions is done by the type system “under the hood.” We present the syntax and static and dynamic semantics of a formal core language and state soundness results. We also illustrate the utility and practicality of our design by using it to express several realistic examples.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Borrowing; D.3.3 [*Programming Languages*]: Language Constructs and Features —Permissions

General Terms Design, Languages, Theory, Verification

Keywords Types, Permissions, Borrowing, Uniqueness, Immutability

1. Introduction

Permissions are annotations on pointer variables that specify how an object may be aliased, and which aliases may read or write to the object [9]. For example, `unique` [20] indicates an unaliased

object, `immutable` [24] indicates an object that can be aliased but cannot be mutated, and `shared` [3] indicates an object that may be aliased and can be mutated. Permission systems have been proposed to address a diversity of software engineering concerns, including encapsulation [24], protocol checking [3, 13], safe concurrency [6, 8], security [5], and memory management [9, 15]. Recently, new programming languages have incorporated permissions [28] or related affine types [26] as fundamental parts of the type system.

In order to leverage permissions in practice, programmers must be able to manipulate them effectively. One form of manipulation is *permission splitting*: for example, converting a unique permission into multiple shared permissions, or alternatively into multiple immutable permissions. A shared (or immutable) permission can then be split further into more shared (respectively immutable) permissions. A second important form of manipulation is *borrowing*: extracting a permission from a variable field, using it temporarily, and then returning all or part of it to the source. For example, a method may require an immutable permission to the receiver; if we have a unique permission to an object, we’d like to call the method on that object, and provided the method does not allow an alias to the receiver to escape, we’d like to get our unique permission back at the end. Crucially, recovering the permission should not require reassigning the original variable (which may not even be assignable).

Borrowing was originally proposed by [19]; however, good support for this feature has remained an open and difficult problem. Prior type-based systems [1, 19, 22] provided a borrowed annotation, but they did not support the immutable references that are an essential part of many recent systems [3, 8, 28]. A number of systems supported borrowing via a program analysis [4, 7], but the program analysis relies on shape analysis, which is fragile and notoriously difficult for programmers to understand. Boyland proposed fractional permissions [8] to support splitting and recombining permissions, with borrowing as a special case, but its mathematical fraction abstraction is unnatural for programmers, to such an extent that the automated tools we know of once again hide the fractions behind an (inscrutable) analysis [3, 17].

A good borrowing facility should have a number of properties. It should support a natural programming style (e.g. avoid awkward constructs like replacing field write with a primitive swap operation [16], or a requirement to thread a reference explicitly from one call to the next by reassigning the reference each time [26]). Reasoning abstractions should likewise be natural (not fractions [8]). It should support borrowing from unique, immutable, and shared variables and fields. Rules should be local so the programmer can understand them and predict how they operate (vs. a non-local analysis [3, 7] or constraint-based inference).

The contribution of this paper is the first borrowing approach that meets all of the above properties. We provide a type system for a Java-like language with permissions that are tracked through local, predictable rules. Our technical approach includes a number of innovations, including change permissions that show the incoming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL’12, January 25–27, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1083-3/12/01...\$10.00

and outgoing permissions to a method parameter, local permissions that modify shared or immutable permissions to denote that they cannot escape, an expressive rule for handling borrowing across conditional branches, and a way to safely restore permissions to the variable or field from which they were borrowed. We formalize our type system, prove it sound, and demonstrate it via a series of examples which can be checked by our prototype implementation.¹

We describe the features of the language in the next section. Section 3 formalizes our type system and gives soundness results. We cover additional related work in section 4, and section 5 concludes.

2. Language Features

In this section we informally explain the features of our language; the next section gives a more formal treatment. Our language is based on Plaid [28]. For our purposes, Plaid is similar to Java, except:

- Instead of classes, Plaid uses states. This is because Plaid has a first-class notion of *typestate*, which can be used to model object state and transitions between states. We don't use typestate in this work, but we adhere to the Plaid syntax.
- Plaid has a `match` construct that evaluates an argument and then uses its type to pick which of several cases to execute.
- Plaid has no `null` value in the surface syntax or programmer-visible semantics. Instead, every object field and variable is initialized to a non-null object.
- Plaid has a first-class notion of permissions. Supporting typestate is one important application of permissions, including the novel borrowing mechanisms discussed in this work.

In the rest of this section we first give an overview of the permissions in our language. Then we explain the mechanisms for creating aliases of variables and fields with consistent permissions. Then we explain the mechanism of change permissions, which allows modular checking of permission flow in and out of methods. Finally, we explain local permissions, which provide a way to split a unique permission into several permissions and later recombine them to unique, without explicit fractions.

2.1 Access Permissions

Permissions are a well-known way of controlling aliasing in applications such as typestate [3] and concurrency control [8]. Our permission system is adapted from the *access permissions* system of [3]. An access permission is a tag on an object reference that says how the reference may be used to access the fields of the object it refers to and how aliases of that reference may be created. In this work, we use the following permissions:

- A `unique` permission to object reference o says that this is the only usable copy of o : if any alias of o exists, then it has permission `none`. The reference may be used to read and write fields of the object O that it points to.
- A `none` permission says that the reference may not be used to read or write the object it points to.
- An `immutable` permission says that the reference o may be used only for reading, and not writing, the fields of the object O that it points to. Further, all other usable (non-`none`) references to O are guaranteed to be `immutable`, so no aliased reference can be used to write O either.

- A `shared` permission says that the reference o may be used for reading and writing, and there is no restriction on aliases of o (so `shared` is like an ordinary reference in Java).

- A `local immutable` or `local shared` permission is like an `immutable` (respectively `shared`) permission, except it can only be passed around in local variables, and can't be assigned to the heap. Local permissions are new with this work and are explained further in Section 2.4.

If any alias of a `unique` reference is created, then the `unique` permission must be consumed (but it can be transferred to the alias). A `unique` permission is therefore like a linear resource [14]. By contrast, `immutable` and `shared` references may be freely replicated.

While other access permissions are possible, these permissions suffice to illustrate the concepts in this paper. Further, these permissions can express a wide range of computation patterns. Other permissions could be added to the system without difficulty.

2.2 Aliasing Variables and Fields

In contrast to a language like Java with unrestricted aliasing, a language with access permissions must maintain careful control over how aliases are created. We now discuss how our language manages permissions for aliases of variables and object fields.

(a) Borrowing Unique from a Variable

```
1 unique x = new S1; // x:unique
2 {
3   unique y = x;    // x:none,y:unique
4 }
```

(b) Borrowing Unique from One of Two Variables

```
1 unique x = new S1; // x:unique
2 unique y = new S2; // x:unique,y:unique
3 {
4   unique z = match(...) {
5     S1=>x; S2=>y; // x:none,y:none,z:unique
6   }
7 }                                // x:unique,y:unique
```

(c) Taking Unique from a Variable

```
1 unique x = new S2; // x:unique
2 unique y = new S1; // x:unique,y:unique
3 {
4   unique z = x;    // x:none,y:unique,z:unique
5   y.fl = z;        // x:none,y:unique,z:none
6 }
```

Figure 1. Examples of variable aliasing.

Aliasing Variables: In our language the following rules govern aliasing of local variables:

1. Our local variables are single-assignment (i.e., they are assigned to only in their declaration) and are declared with explicit permissions, so the needed permission is always available on the left-hand side of a variable assignment.
2. In typing and assignment, we compute all variables and fields such that the value returned by the right-hand side of the assignment may be obtained by reading the variable or field. Because we can't statically resolve which `match` case will be taken, there may be more than one.
3. We maintain a typing environment, which we call a *context*, with the permission associated with each variable. For each possible source variable, we make sure there is enough permission in the context to extract the needed permission. We use permission splitting rules, given formally in the next section, to com-

¹available at <http://code.google.com/p/plaid-lang/>

- pute the permission remaining in the source variables after the extraction, and we update the context accordingly.
- At the end of the scope of the assignee variable, we restore whatever permission is left in that variable to the source variables. For this operation we use permission joining rules, which are the reverse of the splitting rules.

Figure 1 shows examples, where the definitions of states S1 and S2 are as follows:

```
state S1 { unique S1 f1 = new S2; }
state S2 case of S1 { unique S1 f2 = new S3; }
state S3 case of S2 { }
```

Here `case` of is like extends in Java and denotes subclassing. In Figure 1(a), variable `x` is declared `unique` and initialized with a fresh object in line 1. In line 3, variable `y` is created and takes the `unique` permission out of `x`, leaving `none` in `x`. When `y` goes out of scope in line 4, its `unique` permission flows back into `x`. Figure 1(b) is similar, except that in line 5, we don't know which variable (`x` or `y`) will be read from at runtime, so we take permissions out of both, record both as source variables, and restore permissions to both at the end of `z`'s scope in line 7. The `match` statement in lines 4–5 says to evaluate the selector expression (represented as ellipses here) to an object reference `o`, then evaluate the whole expression to `x` if the type of `o` matches `S1`, to `y` if the type of `o` matches `S2`, and to halt if there is no match. While this example is simple, in general typing `match` in our language is subtle and requires *merging* the contexts generated by the different match cases; the details are given in Section 3.2. Figure 1(c) shows an example where the permission read out of `x` into `z` is stored into the heap, so it can't be returned to `x` at the end of `z`'s scope.

(a) Borrowing Unique from a Field

```
1 unique x = new S1; // x:unique
2 {
3     unique y = x.f1; // x:(unique,f1:none),y:unique
4 }
```

(b) Borrowing Unique from a Variable or Field

```
1 unique x = new S1; // x:unique
2 unique y = new S1; // x:unique,y:unique
3 {
4     unique z = match(...) {
5         S1=>x; S2=>y.f1;
6         // x:none,y:(unique,f1:none),z:unique
7     }
8 } // x:unique,y:unique
```

(c) Taking Unique from a Field

```
1 unique x = new S1; // x:unique
2 unique y = new S1; // x:unique,y:unique
3 y.f1 = x.f1; // x:(unique,f1:none),y:unique
```

Figure 2. Examples of field aliasing.

Aliasing Fields: The rules for aliasing of fields are similar to those for aliasing of local variables, with two exceptions. First, pulling a permission out of a field can cause the residual permission to violate the statically declared field permission. For example, pulling `unique` out of a field declared `unique` causes the field to have permission `none`. In this case, we say the field is *unpacked* [12]. If an object has any unpacked fields, then we say the object is unpacked. We must account for the actual permissions when accessing the fields of an unpacked object. Second, since fields can be assigned to, the reference in the field at the point of permission restore may not be the same reference that the permission came from. In this case, we must be careful not to restore permissions to the wrong reference, which would violate soundness.

Unpacking fields: To address the first issue, we store the current permission of each unpacked field of each object in the context. Figure 2 shows examples, where `S1` and `S2` are defined as before. In Figure 2(a), line 3 shows the context after taking a `unique` permission out of `x.f1`. The notation `x: (unique, f1: none)` means that `x` has `unique` permission and points to an unpacked object with `none` permission for field `f1`. Figure 2(b) shows how to use the same mechanism from Figure 1 to pull a permission out of either a variable or a field. Figure 2(c) shows an example of taking a `unique` permission from a field and assigning it to another field.

To ensure soundness, we place three restrictions on field unpacking. First, an object can be unpacked via a variable with `unique` permission, but not immutable or shared permission. This is so outstanding aliases to the object don't become inconsistent.² Our language also allows taking an immutable permission out of a unique field `v.f` given immutable permission to `v`. In this case our type system does not report `v` as unpacked in the context; this is sound because once an object is seen with immutable permission it can never become `unique` again.

Second, once an object `o` stored in variable `v` is unpacked, permission to `o` may not be assigned to another variable or the heap until `v` is packed again; otherwise we would not be able to track the unpacked state of `o` through `v`'s information in the context. For example, the following code is not allowed, because `x` is unpacked when it is assigned to `z`:

```
1 unique x = new S1; // x:unique
2 unique y = x.f1; // x:(unique,f1:none),y:unique
3 unique z = x; // Disallowed because x is not packed
```

Third, a variable `v` must be packed at the point where it goes out of scope; in particular method formal parameters must be packed at the end of a method body. That is because the permission stored in `v` could be flowing back to a different (packed) variable that gave its permission to `v` when `v` was declared.

In practice, the programmer can comply with the latter two requirements by carefully managing the scopes of variables that read permissions from unique fields and/or writing fresh objects into the fields to pack the fields at appropriate points. While in some cases these writes may not be strictly necessary, the requirements do not seem to be onerous in the examples we have studied. These restrictions could be relaxed at the cost of additional language complexity (for example, stating in method signatures which fields of an incoming parameter must be packed).

Consistent permission restore: As an example of the second issue identified above (erroneous permission restore), consider the following code:

```
1 unique x = new S2; // x:unique
2 {
3     unique y = x.f1; // x:(unique,f1:none),y:unique
4     x.f1 = new S2; // x:unique,y:unique
5     x.f2 = x.f1; // x:(unique,f1:none),y:unique
6 }
```

Restoring a permission to `x.f1` at the end of `y`'s scope in line 6 would create an alias between `x.f1` and `x.f2`, both with `unique` permission. The problem is that the reference in field `x.f1` in line 6 is not the same reference to which permission was taken in line 3, so restoring to it would be wrong.

To prevent this from happening, the static typing rules maintain an identifier that is updated every time a field goes from packed to unpacked. The permission restore occurs only if the identifier at the point of the field access matches the identifier at the point of restoration. For example, in the code shown above, at line 3 where

² Local permissions, discussed in Section 2.4, provide an additional way to unpack objects.

`x.f1` is unpacked and its permission read into `y`, an identifier i is associated with `x.f1` and with `y`. Then in line 5, when `x.f1` is unpacked again, `x.f1` gets a fresh identifier i' . In line 6, when y goes out of scope, its identifier i does not match the identifier i' of the source location `x.f1`, so permission is not restored from `y` into the location. Thus the identifier mechanism approximates object identity at runtime; the approximation is conservative because each assignment is assumed to assign a different object reference, even if the same one is actually assigned twice.

2.3 Modular Checking with Change Permissions

To support modular checking of permission flow across method scopes, we introduce a language feature called a *change permission*. Change permissions are inspired by, and similar to, change types in Plaid [28]. However, whereas a change type in Plaid says that an object may transition from one state to another (to support typestate), in our language change permissions specify only that a reference permission changes from a stronger to a weaker permission; the object type always persists. Further, while previous systems can distinguish borrowed and consumed permissions at method boundaries [7, 11], change permissions are more flexible, because they can record a change to any weaker permission (not just none).

Syntactically, a change permission looks like $\pi \gg \pi'$, where π and π' are permissions. Change permissions appear on method formal parameters, including the implicit parameter `this`. For example, a formal parameter can be declared $\pi \gg \pi'$'s `x`, where `s` is a state name. This declaration says the caller must ensure that on entry to the method permission π is available for the reference `o` stored in `x`, while the callee must ensure that on exit from the method permission π' is available for `o`. The bare permission π can also function as a change permission; it is shorthand for $\pi \gg \pi$.

Change permissions naturally support both borrowing and non-borrowing uses of unique permissions. For example, a change permission `unique >> unique` says that the permission in the parameter value must be unique on entry to the method, and the unique permission will be restored at the end of the method; whereas `unique >> none` says that a unique permission is taken and not returned (for example, because it is stored on the heap).

```

1 state Cell {}
2 state Cons case of Cell {
3   immutable Data data;
4   unique Cell next;
5 }
6 state List {
7   unique Cell head = new Cell;
8   void prepend(immutable Data elt) unique {
9     unique Cons newHead = new Cons with {
10       this.data = elt;
11       this.next = this.head; // this:(unique,head:none)
12     }
13     this.head = newHead; // this:unique
14   }
15 }
16
17 unique List list = new List; // list:unique
18 list.prepend(new Data); // list:unique

```

Figure 3. List prepend example.

Figure 3 shows an example using `unique >> unique` (written in shorthand form as `unique`) to update a list with unique links. Line 1 defines a `Cell` state representing an empty list cell. Lines 2–5 define a `Cons` cell which is a substate of `Cell`; it has a `data` field with `immutable` permission and a `next` field with `unique` permission. Lines 6 and following define the actual list. It has a `unique Cell` for its head and a method `prepend` that (1) accepts an `immutable` permission in `elt` and a `unique` permission

in `this` (unique in line 8 represents the change permission associated with `this`); (2) leaves a `unique` permission in `this` on exit; and (3) returns nothing. The comments in lines 11 and 13 show what happens when checking the method body. In line 11, `this` is unpacked when the `unique` permission is taken out of `this.head` and assigned into `newHead`. In line 14, `this` is packed back up when `newHead` is assigned into `this.head`.

Lines 17–18 show how things look from the point of view of a caller of `prepend`. In line 17, `list` gets a fresh reference with `unique` permission. This `unique` permission is passed into `prepend` in line 18. However, because of the change permission `unique >> unique`, the permission is restored on return from the method, and `list` still has `unique` permission at the end of line 18. Note that we could also have restored the permission by returning a reference from `prepend` and assigning it back into `list` but this is awkward and would require assignment into local variables.

```

1 state Data {
2   immutable Data publish() unique>>immutable {
3     // Add timestamp
4     this;
5   }
6   ...
7 }
8
9 unique List list = new List;
10 unique Data data = new Data; // data:unique
11 data.publish(); // data:immutable
12 list.prepend(data); // data:immutable

```

Figure 4. Publication example.

Figure 4 shows another example, this time using a change permission `unique >> immutable`. In this example, the `Data` class has a `publish` method that (1) requires a `unique` permission to `this`; (2) uses the `unique` permission to write a time stamp to the object; then (3) “freezes” the object in an `immutable` state so it can only be read and never written by the rest of the program. Lines 9–12 show how this state might be used. Line 10 creates a fresh `Data` object with `unique` permission. Line 11 calls `publish` on `Data`, changing its permission to `immutable`. Line 12 puts the `immutable` permission into a list. Notice that since `publish` also returns an `immutable` permission to `this`, we could also have written `list.prepend(data.publish())`; but using the change permission on variable `data` makes clear in the client code that the same object is going into `publish` and into `prepend`.

2.4 Local Permissions

An important pattern for access permissions is to divide a `unique` permission up into several weaker permissions, use the weaker permissions for a while, and then put all the permissions back together to reform the original `unique`. Doing this requires careful accounting to ensure there are no outstanding permissions to the reference (other than `none`) at the point where the `unique` is recreated. One way to do this accounting is to use fractions [8]. However, while powerful, fractions are also difficult to use and suffer from modularity problems [17]. Instead, we observe the following:

- The complexity of fraction-based solutions arises in large part because they allow permissions to be stored into the heap, then taken out of the heap and recombined into `unique`.
- A common use of permissions split off from `unique` is to pass them into methods, where they are used in local variables and then returned, i.e., they never go into the heap.

Motivated by these observations, we introduce a new kind of permission called a *local permission*. A local permission is des-

ignated with the keyword `local`, which may modify a shared or `immutable` permission. A local permission annotates a local variable; it says that any aliases of the variable created during its lifetime exist only in local variables and are never stored to the heap. (Local permissions are not necessary for `unique` or `none`, because the permission itself already contains all the information about aliasing to the heap: a `unique` local variable says there is no alias on the heap or anywhere else, and a `none` local variable says it doesn't matter.) Since local permissions exist only in local variables, when all the variables that borrowed local permissions go out of scope, we can reform the original `unique` permission.

Borrowing Local Permissions from Variables: A local permission may be borrowed from a variable with `unique` permission, leaving a special *borrow permission* in the context. The borrow permission is used internally for accounting purposes, but never appears in the programmer-visible language syntax. For example, borrowing `local immutable` from a `unique` variable leaves `borrow(unique, immutable, 1)` in the context for that variable. This entry says we are borrowing `local immutable` permissions from a `unique` permission, and one alias is outstanding. Borrowing again from the same variable increments the counter, while joining decrements the counter. Thus the counter tracks the number of outstanding local aliases to the original `unique` permission; when the counter is 1, joining the last local permission recreates `unique`. Note that we do this counting only when creating local permissions, because in this language `immutable` or `shared` permissions are never joined to `unique`.

(a) Rejoining Local Permissions to Unique

```

1 unique x = new S; // x:unique
2 {
3     local immutable y = x;
4     // x:borrow(unique, immutable, 1), y:local immutable
5     {
6         local immutable z = x;
7         /* x:borrow(unique, immutable, 2),
8          y:local immutable, z:local immutable */
9     }
10    // x:borrow(unique, immutable, 1), y:local immutable
11 }
12 // x:unique

```

(b) This Example Does Not Type Check

```

1 local immutable S escape(unique S x) none {
2     // x:unique
3     local immutable y = x;
4     // x:borrow(unique, immutable, 1), y:local immutable
5     y; // Local immutable permission taken here
6     /* x:borrow(unique, immutable, 1),
7      y:borrow(local immutable, immutable, 1) */
8 }

```

Figure 5. Borrowing local permissions from variables.

Figure 5(a) shows an example. The counter for `x` becomes 1 in line 4 when `y` borrows from it, and 2 in line 7 when `z` borrows from it. When `z` goes out of scope, the counter goes back down to 1, and when `y` goes out of scope, `x` becomes `unique` again. When the counter exceeds one (as in line 7), we don't rejoin to `unique` yet because there are still aliases outstanding. Notice how the counters in the borrow permissions effectively account for fractions of permissions, by counting outstanding aliases. However, these fractions are hidden in the typing and never seen or manipulated by the programmer.

We must carefully account for local permissions that might escape the current scope; otherwise we could not soundly reason that all local permissions are out of scope when the local variables holding them are out of scope. Figure 5(b) shows how we do this. In line 3, `y` takes a local `immutable` permission from `x`,

and line 5 attempts to return the local `immutable` permission to the caller, while retaining the `unique` permission in `x` (as shown in the signature in line 1 — remember that `unique x` is shorthand for `unique>>unique x`). Of course this should not be allowed. This example doesn't type check, because when `y` goes out of scope at the end of the method, two borrow permissions would have to be joined, and our typing rules don't allow this. Only a local permission can be joined with a borrow permission. More generally, all aliases borrowed from a local permission must be rejoined before the local permission can be rejoined to `unique`. However, if the method signature in line 1 said `unique>>none S x`, then the example would type check, because now `x` wouldn't have to be rejoined to `unique` to satisfy the parameter's change permission.

```

1 state Cell { int size() none {0}; }
2 state Cons case of Cell {
3     immutable Data data;
4     unique Cell next;
5     int size() local immutable { 1 + this.next.size(); }
6 }
7 state List {
8     unique Cell head = new Cell;
9     int size() local immutable { this.head.size(); }
10 ...
11 }
12
13 unique List list = new List; // list:unique
14 list.prepend(new Data); // list:unique
15 int size = list.size(); // list:unique

```

Figure 6. List size example.

Borrowing Local Permissions from Fields: Our language also allows borrowing local permissions from `unique` fields. This is particularly useful when the permission to the object is local. Otherwise, we would need `unique` permission to the object even to read a `unique` field of the object, and it is well known that this requirement is very restrictive (essentially, any access to a linear reference must be through a chain of linear references) [13].

Figure 6 shows how this works for a simple example that computes the size of a list with `unique` references, requiring read-only access to the list. In line 15, `size` requires `local immutable` permission to the list (line 11), so local `immutable` permission is borrowed from `list` as discussed above, then put back to reform `unique` at the end of the method call. Inside the `size` method of `List`, in line 9, `this` has local `immutable` permission at the start of the method. At the call of `this.head.size()`, local `immutable` permission is borrowed from `this.head`. At that point, the context entry for `this` is

```
this: (local immutable, head:borrow(unique, immutable, 1))
```

saying that `this` has local `immutable` permission and points to an unpacked object with one local `immutable` reference borrowed from its `head` field. On return from `this.head.size()`, the permissions are put back together to repack the object. The same thing happens in `Cell.size` (line 5). Notice that the ability to borrow local permissions is very useful here: without it, we would either have to permanently convert the `unique` to `immutable`, thereby destroying the `unique` permission to the list to compute its size, or we would have to require `unique` permission to the list for the `size` computation, which is too restrictive.

One subtlety of this mechanism is that it allows multiple local variables that point to the same object to have different information about whether the object is packed. For example, consider an object `O` with a `unique` field `f` and two aliases of `O`, `x` and `y`, where `y` was created as an alias of `x` by pulling a `local σ` from the `unique` permission in `x`. This leaves `x` with the permission `local σ` as

well, which allows us to pull a local σ from $x.f$, unpacking x . Since the type system does not explicitly track what variables are aliases y remains packed. This is fine because the splitting rules prevent anything other than a local σ from being pulled out through $y.f$. Furthermore, by the time y leaves scope and x becomes unique again, all local aliases to the field f of O must have been returned leaving it unique as well. Thus, it will be safe to pull a unique permission from $x.f$ as allowed by the static rules. At runtime, we will know that x and y are in fact aliases and do the proper accounting for the permission in field f even when permission are pulled from the field through different aliases. More details on this mechanism are given in Section 3.

3. Formal Language

In this section we formalize the ideas developed in the previous section. We give a syntax, static semantics, and dynamic semantics for a core language. Then we state the key soundness results, which are proved in our companion technical report [23].

3.1 Syntax

Figure 7 gives the syntax for the core language. A program consists of a number of state declarations S and an expression e to evaluate. A state consists of a state name s , a parent state s' (possibly itself), and field and method declarations. Fields F and methods M are declared in the usual way, except that field types specify permissions π , and method parameters specify change permissions $\pi >> \pi'$. Fields have initializer expressions. The change permission appearing in the method declaration before the method body is the change permission associated with the implicit parameter `this`.

The rest of the language features are standard for an expression-based object-oriented language. The `match` expression evaluates e to an object reference and compares its runtime state s' to the states s named in the cases. It executes the first case for which s' is a subtype of s , with a runtime error if there is no match. $v.f = e$ updates the object in the field $v.f$ to the value of e , which it also returns as the result of the expression. `new s` creates an object of state s and initializes its fields by running the initializer expressions given in the state definition. The sequence expression evaluates each of its subexpressions in order and returns the result of the last one as the value of the whole expression.

Programs	P	$S^* e$
States	S	state s case of $s \{ F^* M^* \}$
Fields	F	$T f = e;$
Methods	M	$T m(\pi >> \pi s x) \pi >> \pi \{ e \}$
Permissions	π	unique none σ local σ
	σ	immutable shared
Types	T	πs
Expressions	e	let $\pi x = e$ in $e e.m(e) v v.f v.f = e$ match $(e) \{(s > e;)^+\} new s \{(e;)^+\}$
Variables	v	this x

Figure 7. Core language syntax. s, f, m , and x are identifiers.

3.2 Static Semantics

We introduce our static semantics by formalizing the notions of states and permissions. We then define the ways that permissions are manipulated and show how the flow of permissions is integrated into the type system.

States: States take the place of standard types in our system. Figure 8 gives the judgments and rules relating to states. $P \vdash s$ says a state is valid if it is declared in P . $P \vdash s \leq s'$ says that s is a substate of s' . Substate relations are defined by the `case of` relation from the state definitions, reflexivity, and transitivity.

$$\begin{array}{c}
 \text{TYPE-STATE} \\
 \boxed{P \vdash s} \quad \frac{\text{state } s \text{ case of } s' \{ F^* M^* \} \in P}{P \vdash s} \\
 \\
 \text{SUBSTATE} \\
 \boxed{P \vdash s \leq s'} \quad \frac{\text{state } s \text{ case of } s' \{ F^* M^* \} \in P}{P \vdash s \leq s'} \\
 \\
 \text{SUBTYPE-REFLEXIVE} \\
 \boxed{P \vdash s \leq s} \quad \frac{}{P \vdash s \leq s} \\
 \\
 \text{SUBSTATE-TRANSITIVE} \\
 \boxed{P \vdash s \leq s' \quad P \vdash s' \leq s''} \quad \frac{}{P \vdash s \leq s''}
 \end{array}$$

Figure 8. Valid states and substates.

$$\begin{array}{c}
 \text{SPLIT-UNIQUE-LOCAL} \\
 \boxed{\pi \Rightarrow \pi' \otimes \pi''} \quad \text{unique} \Rightarrow \text{local } \sigma \otimes \text{borrow(unique, } \sigma, 1) \\
 \\
 \text{SPLIT-LOCAL} \\
 \boxed{\text{local } \sigma \Rightarrow \text{local } \sigma \otimes \text{borrow(local } \sigma, \sigma, 0)} \\
 \\
 \text{SPLIT-LOCAL-INCREMENT} \\
 \boxed{\text{borrow}(\pi, \sigma, n) \Rightarrow \text{local } \sigma \otimes \text{borrow}(\pi, \sigma, n+1)} \quad \text{SPLIT-NONE} \\
 \pi \Rightarrow \text{none} \otimes \pi \\
 \\
 \text{SPLIT-UNIQUE-SYMMETRIC} \\
 \boxed{\text{unique} \Rightarrow \sigma \otimes \sigma} \quad \text{SPLIT-SYMMETRIC} \\
 \sigma \Rightarrow \sigma \otimes \sigma \\
 \\
 \text{SPLIT-SYMMETRIC-LOCAL} \\
 \boxed{\sigma \Rightarrow \text{local } \sigma \otimes \sigma} \quad \text{SPLIT-UNIQUE} \\
 \text{unique} \Rightarrow \text{unique} \otimes \text{none}
 \end{array}$$

Figure 9. Splitting variable permissions.

Permissions: To the set of source permissions we add an additional form needed to track local permissions so they can be joined back to unique:

$$\pi ::= \dots | \text{borrow}(\pi, \sigma, n),$$

where n is a natural number. The `borrow` permission appears only in the context for a local variable or field that has had local permissions split from it. Inside `borrow`, π represents the original permission before the first split (unique or local σ), σ represents the kind of local permission borrowed (immutable or shared), and n counts the number of borrowings.

Splitting variable permissions: The judgment $\pi \Rightarrow \pi' \otimes \pi''$ says that if variable v has permission π , then π' can be taken out of v leaving π'' in v . Figure 9 gives the rules for this judgment. Notice that SPLIT-UNIQUE-SYMMETRIC permanently splits unique into symmetric permissions, making it impossible to ever regain the unique permission. In contrast, the SPLIT-LOCAL-* rules pull a local permission and leave behind a `borrow` permission which will be turned back into the original permission when all the local permissions are returned. In these rules, the count is set to reflect the net gain in local permissions: splitting a local from a unique generates a new local, so the count starts at 1 (SPLIT-UNIQUE-LOCAL); pulling from an existing local replaces the original local with a `borrow`, resulting in no net gain in local permissions, so the count starts at 0 (SPLIT-LOCAL); taking a local from an existing `borrow` creates a new local and thus increments the count (SPLIT-LOCAL-INCREMENT). Also, SPLIT-SYMMETRIC-LOCAL says we may pull local σ out of σ . However, we can't get a non-local permission out of a local, so (in conjunction with the FIELD typing rule) we can't store local permissions to the heap.

$\pi_1 \cdot \pi_2 \Rightarrow \pi_3 \otimes \pi_4$	SPLIT-FIELD-UNIQUE $\frac{\pi \Rightarrow \pi' \otimes \pi''}{\text{unique}.\pi \Rightarrow \pi' \otimes \pi''}$
SPLIT-FIELD-PRESERVE $\pi \neq \text{none} \quad \pi' \Rightarrow \pi'' \otimes \pi'$ $\pi \cdot \pi' \Rightarrow \pi'' \otimes \pi'$	SPLIT-FIELD-UNIQUE-SYMMETRIC $\sigma.\text{unique} \Rightarrow \sigma \otimes \text{unique}$
SPLIT-FIELD-LOCAL $\pi \Rightarrow \text{local } \sigma \otimes \pi'$ $\text{local } \sigma.\pi \Rightarrow \text{local } \sigma \otimes \pi'$	SPLIT-FIELD-BORROW $\frac{\pi' \Rightarrow \text{local } \sigma \otimes \pi''}{\text{borrow}(\pi, \sigma, n).\pi' \Rightarrow \text{local } \sigma \otimes \pi''}$

Figure 10. Splitting field permissions.

$\pi \otimes \pi' \Rightarrow \pi''$	JOIN-NOT-UNIQUE $\frac{\pi'' \neq \text{unique} \quad \pi'' \Rightarrow \pi \otimes \pi'}{\pi \otimes \pi' \Rightarrow \pi''}$
	JOIN-NOT-SYMMETRIC $\frac{\neg \exists \sigma. (\pi = \sigma \wedge \pi' = \sigma)}{\pi \otimes \pi' \Rightarrow \pi''}$

Figure 11. Joining permissions.

Splitting field permissions: The judgment $\pi_1 \cdot \pi_2 \Rightarrow \pi_3 \otimes \pi_4$ says that if variable v has permission π_1 , and field $v.f$ has permission π_2 , then permission π_3 can be taken out of f , leaving π_4 behind. Figure 10 gives the rules. If the variable permission is unique, we can split permissions from the field as if it were a variable (SPLIT-FIELD-UNIQUE), possibly unpacking the field. Otherwise, we allow splitting in three cases. First, if the variable permission is not none, then we can do any splitting allowed for variables that preserves the original field permission (SPLIT-FIELD-PRESERVE). Second, if the variable permission is σ , then we can take permission σ out of a unique field $v.f$ without unpacking the object (SPLIT-FIELD-UNIQUE-SYMMETRIC). This is sound because all other references to the object must have permission σ for the remainder of program execution, and so all other accesses will consistently see the field with permission σ . Third, if the variable permission is local or borrow, then we can take a matching local permission out of a field with unique or borrow permission (SPLIT-FIELD-LOCAL, SPLIT-FIELD-BORROW). This is sound because all local permissions to the object and to the field must go out of scope before a unique permission to the object can be regained. In the mean time, all references to the object and to the field have compatible local or borrow permissions.

Joining permissions: The judgment $\pi \otimes \pi' \Rightarrow \pi''$ says that permissions π and π' may be joined to form permission π'' . The joining rules, given in Figure 11, are very simple: we just reverse all the variable splitting rules, except for SPLIT-UNIQUE-SYMMETRIC, which can't be undone.

Linear Context: To track permission flow, our typing rules use a *linear context*. It is similar to a standard typing environment, except that it maps variables to types that include a permission that may change over the course of typing. The linear context Δ is defined as follows, where i is chosen from an arbitrary set of identifiers:

$$\Delta ::= \emptyset \mid v : (T; \Pi), \Delta \quad \Pi ::= \emptyset \mid f : (\pi, i), \Pi$$

For each variable v in scope, Δ stores a type $T = \pi s$ and a set Π containing the *unpacked state* $f : (\pi', i)$ of the fields f in the state s . A field $f : (\pi', i) \in \Pi$ records two pieces of information. First, π' indicates the current permission in the field. f will only appear in

RESTORE-EMPTY $P; \Delta; \pi \vdash \ell^* : \Delta'$	RESTORE-VAR-ABSENT $\neg \exists T, \Pi. (v : (T; \Pi) \in \Delta) \quad P; \Delta; \pi \vdash \ell^* : \Delta'$ $P; \Delta; \pi \vdash \ell^*, v : \Delta'$
RESTORE-VAR $\Delta = v : (\pi' s; \Pi), \Delta' \quad \pi \otimes \pi' \Rightarrow \pi'' \quad P; v : (\pi'' s; \Pi), \Delta'; \pi \vdash \ell^* : \Delta''$ $P; \Delta; \pi \vdash \ell^*, v : \Delta''$	RESTORE-FIELD-PACKED $v : (T; \Pi) \in \Delta \quad \text{packed}(f, \Pi) \quad P; \Delta; \pi \vdash \ell^* : \Delta'$ $P; \Delta; \pi \vdash \ell^*, (v.f, i) : \Delta'$
RESTORE-FIELD-STALE $v : (T; \Pi, f : (\pi'', j)) \in \Delta \quad i \neq j \quad P; \Delta; \pi \vdash \ell^* : \Delta'$ $P; \Delta; \pi \vdash \ell^*, (v.f, i) : \Delta'$	RESTORE-FIELD-UNPACKED $\Delta = v : (\pi s; \Pi, f : (\pi_2, i)), \Delta' \quad \text{field-type}(P, s, f) = \pi' s' \quad \pi_1 \otimes \pi_2 \Rightarrow \pi_3 \quad \pi_3 \neq \pi' \quad P; v : (\pi s; \Pi, f : (\pi_3, i)), \Delta'; \pi_1 \vdash \ell^* : \Delta''$ $P; \Delta; \pi_1 \vdash \ell^*, (v.f, i) : \Delta''$
RESTORE-FIELD-PACK $\Delta = v : (\pi s; \Pi, f : (\pi_2, i)), \Delta' \quad \text{field-type}(P, s, f) = \pi_3 s' \quad \pi_1 \otimes \pi_2 \Rightarrow \pi_3 \quad P; v : (\pi s; \Pi), \Delta'; \pi_1 \vdash \ell^* : \Delta''$ $P; \Delta; \pi_1 \vdash \ell^*, (v.f, i) : \Delta''$	

Figure 12. Restoring permissions. The predicate $\text{packed}(f, \Pi)$ is true if no binding for f appears in Π . $\text{field-type}(P, s, f) = T$ says that field f is defined in state s with type T in the program P .

the unpacked state of v if π' is distinct from the declared permission of the field f in the state of v . Second, the object identifier i is used to prevent the restoration of permissions from one object to another object. For any $v : (T; \Pi) \in \Delta$ and field f declared in the state of v , if there exists $f : (\pi, i) \in \Pi$, then we say “the field f of v is unpacked”; otherwise we say “the field f of v is packed.” By extension, if $\Pi = \emptyset$, then we say the object pointed to by v (or just v) is packed; otherwise it is unpacked.

Restoring Permissions: At certain points in the typing, permissions flow back into the linear context, such as when a let-bound variable goes out of scope. We restore permissions to a *source location list* consisting of zero or more elements ℓ :

$$\ell ::= v \mid (v.f, i).$$

Each element ℓ stores a location to restore to (either a variable v or a field $v.f$). For fields $v.f$, the identifiers i ensure that we restore permissions only to fields that have not been assigned to since the permission was taken out. We use a list because match expressions may report different source locations for each case. However, we do not allow duplicates in the list.

The judgment for restoring permissions is $P; \Delta; \pi \vdash \ell^* : \Delta'$. It says that if we start with context Δ and join permission π with the permission in each location in ℓ^* , then we get a new context Δ' . Figure 12 gives the rules for this judgment. If the next element in the source location list is a variable appearing in the context, then we restore the permission to it (RESTORE-VAR). Because we do not check that all of the locations in a source location list remain valid when passed outside of a let scope, it is possible that the variable may not be in the context to return to, in which case we do nothing (RESTORE-VAR-ABSENT).

When restoring to a field, there are several cases to consider. If the field is already packed, this means that pulling the permission did not unpack the field (left it unchanged) or the field was reassigned since the permission was pulled, so we do nothing (RESTORE-FIELD-PACKED). If the field is unpacked, and the

$\Delta_3 = \text{merge}(\Delta_1, \Delta_2)$	<p>MERGE-PACKED</p> $\frac{\pi_1 \Rightarrow \pi_3 \otimes \pi_2 \vee \pi_2 = \text{none}}{v : (\pi_2 s; \emptyset) = \text{merge}(v : (\pi_1 s; \emptyset), v : (\pi_2 s; \emptyset))}$	$\Pi \neq \emptyset$ $\frac{\pi_1 \Rightarrow \pi_3 \otimes \pi_2 \vee \pi_2 = \text{none}}{v : (\pi_2 s, \Pi) = \text{merge}(v : (\pi_1 s; \emptyset), v : (\pi_2 s; \Pi))}$
	<p>MERGE-UNPACKED-EQUAL</p> $\frac{v : (\pi_1 s, \Pi_3) = \text{merge}(v : (\pi_1 s; \Pi_1), v : (\pi_1 s; \Pi_2)) \quad \pi_1 \cdot \pi_2 \Rightarrow \pi_4 \otimes \pi_3 \vee \pi_3 = \text{none}}{v : (\pi_1 s; \Pi_3, f : (\pi_3, i)) = \text{merge}(v : (\pi_1 s; \Pi_1, f : (\pi_2, i)), v : (\pi_1 s; \Pi_2, f : (\pi_3, i)))}$	
	<p>MERGE-UNPACKED-UNEQUAL</p> $\frac{v : (\pi_1 s, \Pi_3) = \text{merge}(v : (\pi_1 s; \Pi_1), v : (\pi_1 s; \Pi_2)) \quad \pi_1 \cdot \pi_2 \Rightarrow \pi_4 \otimes \pi_3 \vee \pi_3 = \text{none} \quad i_1 \neq i_2 \quad \text{fresh}(i_3)}{v : (\pi_1 s; \Pi_3, f : (\pi_3, i_3)) = \text{merge}(v : (\pi_1 s; \Pi_1, f : (\pi_2, i_1)), v : (\pi_1 s; \Pi_2, f : (\pi_3, i_2)))}$	

Figure 13. Merging contexts in typing `match` (selected rules).

identifiers in the context and the source location don't match, then we also do nothing (RESTORE-FIELD-STALE). This occurs when an expression in an inner scope assigns to the field and then later unpacks it again, meaning the returned permission represents a permission to a different object than is in the field.

If f is unpacked, and the identifiers match, then there are two cases. First, if restoring the permission doesn't leave a permission equal to the declared permission for f , then we update the unpacked state (RESTORE-FIELD-UNPACKED). Otherwise we pack up the field (RESTORE-FIELD-PACK) by removing it from Π . Notice we don't consider the case where $v.f$ appears in the location list but v doesn't appear in the context. Why not? There are two ways this could happen: either $v.f$ is returned as the value of the `let` expression that declares v , or $v.f$ is returned as the value of a method body, where v is `this` or the method formal parameter. In either case v must be packed after the evaluation of $v.f$ (rules METHOD in Figure 15 and LET in Figure 14). Therefore, $v.f$ can not appear in the source location list (see rule FIELD-ACCESS-PACKED in Figure 14).

Merging Contexts: Each case in a `match` expression may update the permissions in the context in a different way. The `merge` judgment from Figure 13 defines how to combine two contexts into a more general context that can be soundly used to type subsequent expressions regardless of which `case` is actually executed. We provide only the important rules for merging contexts Δ_1 and Δ_2 ; the rest of the rules are about pulling apart the contexts and comparing elements. In summary we use the following rules for merging the two context entries for a variable v :

1. If v is packed in both Δ_1 and Δ_2 (MERGE-PACKED), then we choose the weaker permission. We define π_1 to be *weaker* than π_2 if π_1 is `none` or there exists $\hat{\pi}$ such that $\pi_2 \Rightarrow \hat{\pi} \otimes \pi_1$. If neither permission is weaker than the other, then the contexts are inconsistent and typing fails.
2. If v is unpacked in both contexts, then we require it to have the same permission in both, and we use the field splitting rules to find the weaker of the two field permissions. If the identifier i associated with the unpacked field is the same in both contexts, then we pass it through (MERGE-UNPACKED-EQUAL), but if it is different we generate a fresh identifier (MERGE-UNPACKED-UNEQUAL). This is correct but conservative because it guarantees that no permission restore will occur to the location.
3. If v is unpacked in only one context (MERGE-ONE-UNPACKED), then we use the unpacked element, but we check that the unpacked permission for v is weaker than the packed one. Otherwise, we would have to pull a permission out of an unpacked object, which is not allowed in this language.

Expressions: Permissions are pulled out of the linear context by typing expressions. The judgment $P; \Delta; \pi \vdash e : s; \Delta'; \ell^*$ takes the starting context Δ , a needed permission π , and an expression e

that specifies where π could be pulled from. It produces the state s of the expression, an updated context Δ' , and a source location list ℓ^* that contains all the possible locations in the context that the permission π may have been pulled from. The exact location is not statically known because which branch of a `match` expression is executed is determined at runtime. Figure 14 gives the rules for typing expressions which we now summarize.

Let: We use the permission π' declared for x to type e , obtaining state s , a context Δ' , and a source location list ℓ^* . Then we use the needed permission π of the whole expression to type e' in the context Δ' augmented by the type binding for x . This produces an updated context Δ'' where x is left with permission π'' , and a source location list ℓ^{**} . We require that x be packed because it is going out of scope. We restore π'' to ℓ^* , the locations that the permission for x may have been pulled from, which generates a final context Δ''' that is returned along with the state and source locations from the body.

Match: For a single-case `match`, we type e with a needed permission `none` yielding some state s and an updated context Δ' . We disregard the source location lists since returning `none` is a no-op. We ensure that the state s has a common superstate with the state s_c named in the `case`. This ensures that s_c is a potentially valid state of e . Using Δ' and the needed permission for the entire expression, we type e' to get the state s' and source location list ℓ^* which are reported as the result of the `case`. For a multiple-case `match`, we recursively check the `match` with all but the first `case`. Then we check the `match` with the remaining `case` in isolation using the original context. Finally, we return the the least upper bound of the resulting states, the merged contexts, and the union of the source location lists from the first `case` and the remaining cases.

Method invocation: We type e in the input context with needed permission π_1 yielding a state s_1 and a context Δ_2 .³ Next we look up the method named m in state s_1 . This gives us the permission π_2 required for the argument. We use π_2 to type e' in the context Δ_2 , which yields a state s'_2 and a context Δ_3 . Now we check that s'_2 conforms to the method parameter state s_2 . Then we check that needed permission π can be extracted from the permission π_3 returned by the method. We generate the output context Δ_5 by restoring the output permissions specified for the argument and receiver in the method signature to their respective source location lists. The outgoing source location list is empty because our system does not track what locations the permissions returned from method calls can come from.

Variables: We require that v is packed, because its value may be assigned to another variable or stored on the heap. We also check that the existing permission can be split to give the needed permis-

³ π_1 is the needed permission for the receiver as determined by the signature of the method m in the state s_1 of the expression e . This permission can be determined by a pre-pass that ignores permissions and just gathers state information for all variables.

$$\begin{array}{c}
\text{LET} \\
\frac{P; \Delta; \pi' \vdash e : s; \Delta'; \ell^* \quad P; \Delta', x : (\pi' s; \emptyset); \pi \vdash e' : s'; \Delta'', x : (\pi'' s; \emptyset); \ell'^* \quad P; \Delta''; \pi'' \vdash \ell^* : \Delta'''}{P; \Delta; \pi \vdash \text{let } \pi' x = e \text{ in } e' : s'; \Delta'''; \ell'^*}
\end{array}$$

$$\begin{array}{c}
\text{MATCH-SINGLE} \\
\frac{\begin{array}{c} P; \Delta; \text{none} \vdash e : s; \Delta'; \ell^* \\ \exists s_l. P \vdash s_l = \text{lub}(s, s_c) \quad P; \Delta'; \pi \vdash e' : s'; \Delta''; \ell'^* \end{array}}{P; \Delta; \pi \vdash \text{match}(e) \{ s_c \Rightarrow e' \} : s'; \Delta''; \ell'^*}
\end{array}
\qquad
\begin{array}{c}
\text{MATCH-MULTIPLE} \\
\frac{\begin{array}{c} P; \Delta; \pi \vdash \text{match}(e) \{ (s_c \Rightarrow e_1)^+ \} : s_1; \Delta_1; \ell_1^* \\ P; \Delta; \pi \vdash \text{match}(e) \{ s_c \Rightarrow e_2 \} : s_2; \Delta_2; \ell_2^* \\ P \vdash s_3 = \text{lub}(s_1, s_2) \quad \Delta_3 = \text{merge}(\Delta_1, \Delta_2) \quad \ell_3^* = \ell_1^* \cup \ell_2^* \end{array}}{P; \Delta; \pi \vdash \text{match}(e) \{ (s_c \Rightarrow e_1)^+ \} : s_3; \Delta_3; \ell_3^*}
\end{array}$$

$$\begin{array}{c}
\text{INVOKE} \\
\frac{\begin{array}{c} P; \Delta_1; \pi_1 \vdash e : s_1; \Delta_2; \ell^* \\ \text{method}(P, s_1, m) = \pi_3 s_3 \ m(\pi_2 \gg \pi'_2 s_2 \ x) \ \pi_1 \gg \pi'_1 \{ e'' \} \quad P; \Delta_2; \pi_2 \vdash e' : s'_2; \Delta_3; \ell'^* \\ P \vdash s'_2 \leq s_2 \quad \pi_3 \Rightarrow \pi \otimes \pi' \quad P; \Delta_3; \pi'_2 \vdash \ell'^* : \Delta_4 \quad P; \Delta_4; \pi'_1 \vdash \ell^* : \Delta_5 \end{array}}{P; \Delta_1; \pi \vdash e.m(e') : s_3; \Delta_5; \emptyset}
\end{array}
\qquad
\begin{array}{c}
\text{VAR} \\
\frac{\begin{array}{c} \Delta = \Delta', v : (\pi' s; \emptyset) \\ \pi' \Rightarrow \pi \otimes \pi'' \quad \Delta'' = \Delta', v : (\pi'' s; \emptyset) \end{array}}{P; \Delta; \pi \vdash v : s; \Delta''; v}
\end{array}$$

$$\begin{array}{c}
\text{FIELD-ACCESS-PACKED} \\
\frac{\begin{array}{c} v : (\pi' s; \Pi) \in \Delta \quad \text{packed}(f, \Pi) \\ \text{field-type}(P, s, f) = \pi'' s' \quad \pi'.\pi'' \Rightarrow \pi \otimes \pi'' \end{array}}{P; \Delta; \pi \vdash v.f : s'; \Delta; \emptyset}
\end{array}
\qquad
\begin{array}{c}
\text{FIELD-ACCESS-UNPACK} \\
\frac{\begin{array}{c} \Delta = \Delta', v : (\pi_1 s; \Pi) \in \Delta \quad \text{packed}(f, \Pi) \quad \text{field-type}(P, s, f) = \pi_2 s' \\ \pi_1.\pi_2 \Rightarrow \pi_3 \otimes \pi_4 \quad \pi_2 \neq \pi_4 \quad \text{fresh}(i) \quad \Delta'' = \Delta', v : (\pi_1 s; \Pi, f : (\pi_4, i)) \end{array}}{P; \Delta; \pi_3 \vdash v.f : s'; \Delta''; (v.f, i)}
\end{array}$$

$$\begin{array}{c}
\text{FIELD-ASSIGN-UNPACKED} \\
\frac{\begin{array}{c} v : (\pi_1 s_1; \Pi) \in \Delta \quad \text{field-type}(P, s_1, f) = \pi_2 s_2 \quad P; \Delta; \pi_2 \vdash e : s_3; \Delta_1; \ell^* \\ \Delta_1 = \Delta_2, v : (\pi_3 s_1; \Pi', f : (\pi_4, i)) \quad \text{assignable}(\pi_3) \quad P \vdash s_3 \leq s_2 \quad \pi_2 \Rightarrow \pi \otimes \pi_2 \quad \Delta_3 = \Delta_2, v : (\pi_3 s_1; \Pi') \end{array}}{P; \Delta; \pi \vdash v.f : e : s_3; \Delta_3; \emptyset}
\end{array}
\qquad
\begin{array}{c}
\text{FIELD-ASSIGN-PACKED} \\
\frac{\begin{array}{c} v : (\pi_1 s_1; \Pi) \in \Delta \quad \text{field-type}(P, s_1, f) = \pi_2 s_2 \\ P; \Delta; \pi_2 \vdash e : s_3; \Delta'; \ell'^* \quad v : (\pi_3 s_1; \Pi') \in \Delta' \quad \text{assignable}(\pi_3) \\ \text{packed}(f, \Pi') \quad P \vdash s_3 \leq s_2 \quad \pi_2 \Rightarrow \pi \otimes \pi_2 \quad \Delta''' = \Delta', v : (\pi_3 s_1; \Pi') \end{array}}{P; \Delta; \pi \vdash v.f : e : s_3; \Delta'''; \emptyset}
\end{array}$$

$$\begin{array}{c}
\text{SEQUENCE-SINGLE} \\
\frac{P; \Delta; \pi \vdash e : s; \Delta'; \ell^*}{P; \Delta; \pi \vdash \{ e; \} : s; \Delta'; \ell^*}
\end{array}
\qquad
\begin{array}{c}
\text{SEQUENCE-MULTIPLE} \\
\frac{\begin{array}{c} P; \Delta; \text{none} \vdash \{(e_i)^+\} : s; \Delta'; \ell^* \quad P; \Delta'; \pi \vdash \{ e' \} : s'; \Delta'', \ell'^* \\ P; \Delta; \pi \vdash \{(e_i)^+ e'\} : s'; \Delta''; \ell'^* \end{array}}{P; \Delta; \pi \vdash \{ (e_i)^+ e' \} : s'; \Delta''; \ell'^*}
\end{array}$$

Figure 14. Typing expressions. $P \vdash s = \text{lub}(s', s'')$ means s is the least state that is a superstate of both s' and s'' . $\text{method}(P, s, m) = M$ means that M is the method named m defined in state s for program P , and $\text{field-type}(P, s, f) = T$ similarly produces the type T of field f from s in P . The predicate $\text{assignable}(\pi)$ holds if π is not none, immutable, local immutable, or borrow(local immutable, immutable, n). $\text{packed}(f, \Pi)$ means $\neg \exists(\pi, i). (f : (\pi, i) \in \Pi)$.

sion, and we leave the residue in the type reported in the outgoing context. The source expression list contains only the variable itself. *Field access:* We start by finding the type of v in the context and then get the defined type of field f in its state s . Since we are only accessing the field of v , we do not update the permission of v . However, we may need to update the unpacked state of f in v . If f starts packed in v 's type, then there are two cases to handle. First, if we can take the required permission out of the field and leave the same permission behind, then we leave the object packed (FIELD-ACCESS-PACKED). The source expression list is empty because no permission needs to be restored to a field that is packed. Second, if we need to leave a different residual permission, then we unpack the field, leaving the residual permission behind (FIELD-ACCESS-UNPACK). We also generate a fresh identifier i and report $(v.f, i)$ as the source location list. If f is already unpacked in v , then we split the needed permission from the current field permission, and replace the current permission in the unpacked field state with the residual permission (FIELD-ACCESS-UNPACKED). We report $(v.f, i)$ as the source location, using the existing identifier i . *Field assignment:* If $v.f$ is packed, then to assign e to it we (1) look in the context to get the permission we need for the field; (2) type e ; (3) check that we have writable permission to v in the resulting context; and (4) check that the states match (FIELD-ASSIGN-PACKED). We also ensure that the permission we need can be split off from the permission needed by the field, while retaining

the field permission. If $v.f$ is unpacked, then we do the same thing, but we pack up f at the end (FIELD-ASSIGN-UNPACKED). In both cases, we do not need to return permissions to the source locations of e because what's left after assigning to $v.f$ must be none or symmetric and returning either is a no-op. For the same reason, the returned source location list is empty.

Object creation and expression sequence: These rules are straightforward. In SEQUENCE-MULTIPLE, notice that we pull the needed permission only for the last element in the sequence, whose value is returned by evaluating the expression; we pull none from the rest of the expressions in the sequence. For example, if x is unique, it is legal to request a unique permission from the expression $\{x; x; x; \}$, because unique is only pulled from the last x . Since returning a none permission is a no-op, we can safely discard the source location lists for these expressions.

Top-Level Program Structure: Figure 15 gives the rules for typing programs, states, fields, and methods. In rule PROGRAM we type the main expression with a needed permission none, because no permission to the result is needed after program execution is complete. In rule FIELD we type the initializer expression in the empty environment to ensure that this isn't stored to the heap by a new expression. Also, we require that user-declared field types cannot be declared with local permissions (borrow permissions are also excluded because they cannot appear in the source). This restriction ensures that a local permission is never assigned to the

$\frac{\begin{array}{c} \text{PROGRAM} \\ P = S^* e \quad \forall S \in S^*. (P \vdash S) \\ P; \emptyset; \text{none} \vdash e : s; \emptyset; \ell^* \end{array}}{\vdash P}$	$\frac{\text{FIELD} \quad T = \pi s \quad P \vdash s \quad \neg \exists \sigma. (\pi = \text{local } \sigma)}{P; \emptyset; \pi \vdash e : s'; \emptyset; \ell^* \quad P \vdash s' \leq s}$	$\frac{}{P \vdash F}$	$\frac{}{P \vdash S}$	$\frac{\text{STATE} \quad \text{state } s' \text{ case of } s'' \{ F'^* M'^* \} \in P \quad \forall F \in F^*. (P \vdash F) \quad \forall M \in M^*. (P, s \vdash M)}{P \vdash \text{state } s \text{ case of } s' \{ F^* M^* \}}$	$\frac{\text{METHOD} \quad \begin{array}{c} P \vdash s \quad P \vdash s_2 \\ \Delta = \text{this} : (\pi_1 s_1; \emptyset), x : (\pi_2 s_2; \emptyset) \quad P; \Delta; \pi \vdash e : s', \Delta'; \ell^* \\ \text{this} : (\pi'_1 s_1; \emptyset), x : (\pi'_2 s_2; \emptyset) \in \Delta' \quad P \vdash s' \leq s \end{array}}{P, s_1 \vdash \pi s \ m(\pi_2 >> \pi'_2 s_2 x) \ \pi_1 >> \pi'_1 \{ e \}}$
--	---	-----------------------	-----------------------	--	---

Figure 15. Programs, states, fields, and methods.

heap. In rule METHOD we check that the return type and parameters are valid. We check that the method body types in the context created by binding the method receiver and parameter to their types and that the resulting state is a substate of the return state. We require that the parameter and receiver be packed in the the output context Δ' and also that their permissions in Δ' agree with the output permissions given by their change types.

For simplicity, we implicitly disallow overriding, shadowing, and/or overloading of fields and methods. We also disallow cycles in the inheritance hierarchy and treat the set of states as a forest of trees, where a state s is a root if it has itself as a parent.

3.3 Dynamic Semantics

An execution state $H; e$ includes a heap H and an expression e . **Heap:** Our model of the heap H is a partial function from object references to object identifiers and from object identifiers to objects. The addition of *object references* provide an additional level of indirection that allows us to explicitly track and reason about different aliases to a single object as in [28]. These serve a formal purpose only and are not necessary in an implementation. Consistent with this indirection, the fields of an object dF^* map field names to object references. Every object reference in $\text{dom}(H)$ maps to an object identifier except the special reference `null`, which is used during object initialization.

$$\begin{aligned} H &::= \text{null} \mid o \mapsto O, H \mid O \mapsto s \{ dF^* \}, H \\ dF &::= f \mapsto o \end{aligned}$$

Expressions: We enhance our expression language to support partially-executed programs. We add object references o as well as the new form `alias(o)` as expressions. `alias(o)` gives computational meaning to splitting a permission of o by creating a new alias to hold the split permission. As alias expressions will be substituted for bound variables in let bodies, we allow them to appear wherever variables can. Finally, we add a partial let form that keeps track of the scope in which an object reference o is bound.

$$e ::= \dots \mid o \mid \text{alias}(o) \mid \text{alias}(o).f \mid \text{alias}(o).f = e \mid \text{let } o \text{ in } e$$

Reduction Rules: We formalize program execution using a small step operational semantics. Given a program P , the reduction rules shown in Figure 16 take one execution state to another.

Congruence: We specify reduction of subexpressions using an evaluation context E defined below:

$$E ::= \square \mid \text{let } \pi \ x = E \text{ in } e \mid \text{let } o \text{ in } E \mid \text{match } (E) \{(s => e;)^*\} \mid E.m(e) \mid o.m(E) \mid \text{alias}(o).f = E \mid \{E; (e;)^*\}$$

As usual, \square is a “hole” that is filled in with the subexpression under evaluation to construct the entire expression being evaluated.

Alias: Reducing an `alias(o)` expression creates a fresh object reference that points to the same object identifier as the original object reference.

Let: Once the bound expression is reduced to an object reference o we substitute `alias(o)` for all occurrences of the bound variable x in the body. Thus, any use of x in the body must create a fresh alias. We also keep track of the scope of o by reducing to the partial let form. Once the body of a partial let has been evaluated to an object reference we remove the scoping annotation.

Match: Once the selector expression becomes an object reference o , we select the first match case, if any, that is a superstate of the state of o . If there is no match, then execution gets stuck.

Method invocation: Once we have object references for the receiver and argument of the method, we substitute the method body surrounded by let expressions binding the receiver and argument. This allows let reduction to handle the substitution and scoping.

Object creation: We create a new object identifier O and give its fields the reference `null`, which will be replaced by the proper values after running the initializer expressions. We reduce to a sequence expression which initializes each field of the object in turn through a fresh object reference o . The final expression in the sequence returns o .

Field access and assignment: A field access evaluates to a fresh alias to the object reference the heap associates with the field. Field assignment replaces the object reference in the field with object reference from evaluating the right-hand side. It returns a fresh alias of this same reference. Since we do not use the target of the field read or assignment except to access its field, we do not need a new alias to it, so we leave the `alias(o)` in the target unevaluated.

Sequence: Each expression in the sequence is evaluated in order until there are no further expressions at which point the sequence is reduced to the object reference resulting from the reduction of the final expression.

3.4 Soundness Results

In this section, we present the main soundness results and supporting definitions for our system. Our technical report [23] contains the full definitions and proofs.

Typing of Dynamic Expressions: We state and prove soundness in terms of a standard dynamic typing. To do this, we need rules for typing object references, `alias` expressions, and partial lets.

First, we enhance the definition of the linear context to map object references to types. We also add entries of the form $\ell \Leftarrow o$ to the context which indicate that o returns its permission to the source location ℓ when it leaves scope. We will also want to be able to return permissions to object references, so we extend the definition of source locations to include them.

$$\begin{aligned} \Delta &::= \dots \mid o : (\pi s, \Pi), \Delta \mid \ell \Leftarrow o, \Delta \\ \ell &::= \dots \mid o \mid (o, f, i) \end{aligned}$$

$\frac{P \vdash H; e \rightarrow H'; e'}{P \vdash H; E[e] \rightarrow H'; E[e]}$	E-CONGRUENCE $\frac{P \vdash H; e \rightarrow H'; e'}{P \vdash H; E[e] \rightarrow H'; E[e]}$	E-ALIAS $\frac{H = H', o \mapsto O \quad o' \notin \text{dom}(H) \quad H'' = H, o' \mapsto O}{P \vdash H; \text{alias}(o) \rightarrow H''; o'}$
E-LET-BINDING $\frac{P \vdash H; \text{let } \pi x = o \text{ in } e \rightarrow H; \text{let } o \text{ in } e[x \leftarrow \text{alias}(o)]}{P \vdash H; \text{let } \pi x = o \text{ in } e \rightarrow H; \text{let } o \text{ in } e}$		E-LET-BODY $\frac{H = H', o \mapsto O}{P \vdash H; \text{let } o \text{ in } o' \rightarrow H'; o'}$
E-MATCH $\frac{\begin{array}{c} O \mapsto s\{dF^*\} \in H' \\ H = H', o \mapsto O \\ P \vdash s \leq s_j \quad \neg \exists k \in [1, j-1]. (P \vdash s \leq s_k) \end{array}}{P \vdash H; \text{match } o \{ (s_i \mapsto e_i)_{i \in [1, n]} \} \rightarrow H'; e_j}$		E-INVOKE $\frac{\begin{array}{c} H = H', o \mapsto O, O \mapsto s\{dF^*\} \\ \text{method}(P, s, m) = \pi_3 s m (\pi_2 >> \pi'_2 s' x) \quad \pi_1 >> \pi'_1 \{ e \} \\ P \vdash H; o.m(o') \rightarrow H; \text{let } \pi_1 \text{ this}=o \text{ in let } \pi_2 x=o' \text{ in } e \end{array}}{P \vdash H; o.m(o') \rightarrow H; \text{let } \pi_1 \text{ this}=o \text{ in let } \pi_2 x=o' \text{ in } e}$
E-FIELD $\frac{\begin{array}{c} H = H', o \mapsto O, O \mapsto s\{dF, f \mapsto o'\}, o' \mapsto O' \\ o' \notin \text{dom}(H) \quad H'' = H, o'' \mapsto O' \end{array}}{P \vdash H; \text{alias}(o).f \rightarrow H''; o''}$	E-ASSIGN $\frac{\begin{array}{c} H; \text{alias}(o') \rightarrow H'; o'' \\ H' = H'', o \mapsto O, O \mapsto s\{dF^*, f \mapsto o''\} \\ H''' = H'', o \mapsto O, O \mapsto s\{dF^*, f \mapsto o'\} \end{array}}{P \vdash H; \text{alias}(o).f=o' \rightarrow H'''; o''}$	
E-NEW $\frac{\begin{array}{c} \text{state } s \text{ case of } s' \{ (\pi_i s_i f_i = e_i)_{i \in [1, n]} M^* \} \in P \\ o, O \notin \text{dom}(H) \quad H' = H, o \mapsto O, O \mapsto s\{(f_i \mapsto \text{null})_{i \in [1, n]}\} \end{array}}{P \vdash H; \text{new } s \rightarrow H'; \{(\text{alias}(o).f_i = e_i)_{i \in [1, n]} o; \}}$	E-SEQUENCE-SINGLE $\frac{P \vdash H; \{o; \} \rightarrow H; o}{P \vdash H; \{o; (e;)^+\} \rightarrow H'; \{(e;)^+\}}$	E-SEQUENCE-MULTIPLE $\frac{H = H', o \mapsto O}{P \vdash H; \{o; (e;)^+\} \rightarrow H'; \{(e;)^+\}}$

Figure 16. Reduction rules.

ALIAS $\frac{\Delta = \Delta', o : (\pi' s; \emptyset) \quad \pi' \Rightarrow \pi \otimes \pi'' \quad \Delta'' = \Delta', o : (\pi'' s; \emptyset)}{P; \Delta; \pi \vdash \text{alias}(o) : s; \Delta''; o}$	OBJECT-REF $\frac{\Delta = \Delta', o : (\pi' s; \emptyset), \ell \Leftarrow o \quad \pi' \Rightarrow \pi \otimes \pi'' \quad \Delta'' = \Delta', \ell \Leftarrow o}{P; \Delta; \pi \vdash o : s; \Delta''; \ell}$
ALIAS-FIELD-ACCESS $\frac{x \notin \text{dom}(\Delta) \quad P; \Delta; x : (\pi' s', \Pi'); \pi \vdash x.f : s; \Delta, x : (\pi' s', \Pi''); \ell^*}{P; \Delta, o : (\pi' s', \Pi'); \pi \vdash \text{alias}(o).f : s; \Delta, o : (\pi' s', \Pi''); \ell^*[x \leftarrow o]}$	ALIAS-FIELD-ASSIGN $\frac{x \notin \text{dom}(\Delta) \quad P; \Delta; x : (\pi' s', \Pi'); \pi \vdash x.f = e : s; \Delta', x : (\pi'' s', \Pi''); \emptyset}{P; \Delta, o : (\pi' s', \Pi'); \pi \vdash \text{alias}(o).f = e : s; \Delta', o : (\pi'' s', \Pi''); \emptyset}$
LET-PARTIAL $\frac{\Delta = \Delta', o : (\pi' s', \Pi'), \ell \Leftarrow o \quad P; \Delta; \pi \vdash e : \pi s; \Delta'', o : (\pi'' s', \emptyset), \ell \Leftarrow o; \ell^*}{P; \Delta; \pi \vdash \text{let } o \text{ in } e : s; \Delta'''; \ell^*}$	$P; \Delta'', \pi'' \vdash \ell : \Delta'''$

Figure 17. Intermediate typing rules.

The rules for typing partial expressions are found in Figure 17. The ALIAS rule says that typing $\text{alias}(o)$ proceeds by typing o like a variable. In particular, the source location is o . A raw object reference o is typed using rule OBJECT-REF which is also similar to the VAR rule with two important exceptions. First, we look in the context to find the return location specified for o . This is because we need to maintain the typing after an alias expression steps to an object reference. In particular, after $\text{alias}(o)$ steps to o' , typing o' should still return o as the source location. Second, the binding for the type of o is removed from the outgoing context to ensure that o is not used later in the program which would break our model of execution where each variable and field is tracked as a separate object reference. To type a field read or assignment after an alias has been substituted for the target variable ALIAS-FIELD* undoes the substitution with a fresh variable x that replaces $\text{alias}(o)$. We give x the type of o from the context when typing the updated expression and then return the resulting outputs with o put back for x in the context and source location list. LET-PARTIAL assumes the scoped object reference o is already in the context along with a return location for it. After the body is typed, the remaining permission to o is restored to its return location.

Soundness Judgments: Figure 18 shows the soundness judgments for the system. $P; \Delta; \pi \vdash H; e : \pi s; \hat{\Delta}; \ell^*$ is the top-level judgment and says that the runtime environment $H; e$ is well-typed with respect to a program P , a context Δ , and a permission π .

The premises of this judgment are as follows. First, the context and the heap must map the same set of object references. Second, the incoming context must be able to be partitioned into two distinct parts: Δ_e to type the expression (which produces the outputs of the judgment), and Δ_f which contains all of the object references stored in fields f in H . Third, three invariants must hold for the context and the heap: (1) consistent object permissions, (2) consistent object references, and (3) distinct fields.

Consistent object permissions: The key soundness condition of our system is that for each O , the set of references o that point to it must have *consistent* permissions in Δ . This condition captures the meaning of the permissions in our system. Figure 19 defines this consistency condition. The judgment $\text{perms}(\Delta, H, O) = [\pi^*]$ forms the list of all the permissions π given to object references o that point to O in H . The judgment $[\pi^*]$ **consistent** places three requirements on this list: (1) if `unique` appears in the list, then it appears only once, and all other permissions are `none`; (2) if a `borrow(unique, \sigma, n)` permission appears in the list, then the number of `local \sigma` permissions must match the sum of the counts of all `borrow` permissions in the list; and (3) no list may contain both `shared` and `immutable` permissions (including the `local` and `borrow` versions).

Consistent object references: Object references must also be internally consistent. We say that $P; \Delta; H \vdash o \text{ ok}$ if the representation of the object reference o in H is consistent with the type of o in

Figure 18. Soundness judgments. $\text{dyn-field-type}(P, s, f, \Pi)$ reports the permission for f in Π if it is recorded there, otherwise the declared permission of f in state s . The set validFieldPerms excludes permissions of the form `local` σ and `borrow(local` $\sigma, \sigma, n)$.

Figure 19. Permission lists and consistent permissions. The notation $L = [\pi^*]$ means that L is a list of permissions (with duplicates allowed). The symbol $++$ denotes list concatenation.

Δ . For the representation to be valid we first need the actual states of the object and its fields to be substates of the states given by the type of o . Second, we do not allow a permission `local` σ or `borrow(local` $\sigma, \sigma, n)$ to appear as the permission in a field or as the permission to an object reference that represents a field. This maintains our invariant that `local` permissions never appear in the heap. Finally, we require that any permission that can be pulled from a field f of the state s through o can also be pulled from the permission of the object reference that represents the field in H . Given the field type $\pi_o . \pi_f$ of f in o , we say that π_r fulfills $\pi_o . \pi_f$ (represented by $\pi_r \triangleright \pi_o . \pi_f$) if any permission that can be split from $\pi_o . \pi_f$ can also be split from π_r . Both judgments are formalized in Figure 18.

Distinct fields: In order to carry out the typing of a partially evaluated expression we must ensure that no object reference o ever appears (1) both in a field and in the expression being typed or (2) in two different fields in H . Otherwise, typing one location could

Figure 20. Weaker contexts. Meta-variable p consists of v or o and $\text{fields}(P, s)$ lists the names of the fields in state s .

have a non-local impact on the permission in the other location that would be difficult to track. This invariant is captured formally by the judgment $\Delta \vdash H \text{ distinct fields}$ in Figure 18.

Weaker Contexts: As an expression is evaluated, specific paths of execution are chosen, reducing the number of possible future execution states. Consequently, the context produced by typing the updated expression gives a more precise view of the states and permissions of locations in the program when compared to the context produced by typing the original expression. For instance, if stepping from e to e' reduces away a `match` expression, then there may

be locations that have permissions pulled from them when typing e but not when typing e' . This leaves more permissions in the resulting context. We introduce the *weaker* relationship between contexts to account for this increase in permissions.

The judgment $P \vdash \Delta_1 \downarrow \Delta_2$ shown in Figure 20 states that Δ_2 is weaker than Δ_1 . The judgment holds if we can transform Δ_1 into Δ_2 by making specific changes to the state and permission at each location (variable or field) appearing in both Δ_1 and Δ_2 . We allow the type of a location in Δ_1 to be replaced in Δ_2 with a type containing a superstate and a weaker permission ($\pi \downarrow \pi$). `none` is weaker than any permission (PERM-WEAKER-NONE). Otherwise, for permission π_2 to be weaker than π_1 , there must exist a single permission $\hat{\pi}$ that can be pulled from π_1 such that the residue is π_2 (PERM-WEAKER). We also admit the special case where π_1 is `unique` and π_2 is a `borrow` permission (PERM-WEAKER-BORROW). This can occur, for instance, if a field with a `borrow` permission is reassigned in the chosen branch of a `match`.

Progress and Preservation: We use these definitions to state the soundness of our system via standard progress and preservation theorems.

Theorem 3.1 (Progress). *If $P; \Delta; \pi \vdash H; e : s; \hat{\Delta}; \ell^*$, then either $e = o$ or $P \vdash H; e \rightarrow H'; e'$ or execution is stuck at a match statement where the matched expression has been evaluated to an object, but there is no matching case.*

Theorem 3.2 (Preservation). *If $P; \Delta; \pi \vdash H; e : s; \hat{\Delta}; \ell^*$, and $P \vdash H; e \rightarrow H'; e'$, then there exists Δ' such that*

1. $P; \Delta'; \pi \vdash H'; e' : s'; \hat{\Delta}'; \ell'^*$ with $P \vdash \hat{\Delta}' \downarrow \hat{\Delta}$ and $P \vdash s' \leq s$
2. $P; \hat{\Delta}; \pi \vdash (\ell^* \setminus \ell'^*) : \hat{\Delta}_r$ where $P \vdash \hat{\Delta}' \downarrow \hat{\Delta}_r$

The first condition of preservation requires that the new environment after reduction is still well-formed, with the extra restriction that the context $\hat{\Delta}$ generated by typing the old expression e must be weaker than the context $\hat{\Delta}'$ produced by typing the updated expression e' . However, this is not strong enough because it does not account for the permissions that were pulled out as a part of typechecking which may be returned to the context later. The second condition provides the extra power we need. Consider the source location list $\ell^* \setminus \ell'^*$ representing all the source locations from which the needed permission π was pulled when typing e but that were not used to get π when typing e' . There is potentially more permission at these locations in $\hat{\Delta}'$ than in $\hat{\Delta}$ because π was not split from them during the typing of e' . However, are we guaranteed that if we restore the pulled permission to these locations in $\hat{\Delta}$ using the judgment defined in Figure 12, the resulting context $\hat{\Delta}_r$ is still weaker than $\hat{\Delta}'$? The second condition of preservation says, “Yes.” This fact is necessary to prove the weakening condition on $\hat{\Delta}'$ from the first condition in some cases such as when E-CONGRUENCE is used to reduce a `let` expression. This and other details of the proof of safety are discussed in our accompanying technical report [23].

4. Related Work

Wadler first introduced the concept of temporarily converting a linear (`unique`) reference into a non-linear reference using the `let!` construct [27]. Other early uniqueness type systems built on his work and added support for borrowing as a special annotation, such as “borrowed” or “lent” [1, 19, 22]. While convenient, these systems did not support borrowing immutable pointers, and generally provided weak guarantees: multiple borrowed pointers could co-exist and interfere with one another. Boyland devised alias burying to address this issue, using shape analysis to ensure that whenever a unique variable was read, all aliases to it were dead

(or “buried”) [7]. While this approach works well in an analysis tool, it is inappropriate for a type system: programmers would have to understand a shape analysis to comprehend and fix a type error message. The authors of Plural [3], which also uses analysis to support borrowing, have observed this to be a problem in practice. In contrast, our system provides a more natural abstraction for reasoning by modeling the flow of permissions through locations in the source.

Boyland proposed *fractional permissions* as a generalization of borrowing that does not require a stack discipline for creating and destroying borrowed aliases [8]. Although fractions have received a lot of attention in the verification community, we know of no practical tool support that leverages fractions—possibly because programmers find fractions an unintuitive abstraction. Instead, tools like Plural [3], Chalice [17], and VeriFast [21] provide abstractions (including borrowing) that hide fractions from users, but the use of program analysis and theorem provers makes these systems less predictable and more difficult to understand than the type system presented here. Boyland and Retert later developed a type system that allows borrowing unique permissions [10]. Like our system, their system tracks permissions taken out of individual fields using a technique they call “carving.” However, where they use a substructural logic for tracking permissions, we use a more predictable linear context to type individual variables.

One of the authors previously observed the importance of borrowing for tracking permissions and presented the first fraction-free permission type system we are aware of that supports borrowing unique and immutable permissions [2]. While the technical details are somewhat different, the system presented in [2], similar to this system, avoids fractions by counting split-off permissions in variable types. We propose `local` permissions to distinguish borrowed permissions syntactically, as well as `none` permissions, both of which remain implicit in [2]. Our system additionally supports share permissions, match expressions and sequences, and our system tracks permissions taken out of individual fields. Unlike [2], we provide a dynamic semantics and prove our system sound.

Other programming languages have incorporated the ideas of uniqueness and borrowing into their type systems but use less flexible or more complicated mechanisms. The Clean programming language [25] is a functional language with support for unique references. However, since the language is functional, there is no concept of returning permissions to a location as in our system.

The Vault programming language [13] allows linear (`unique`) references to be split into guarded (`immutable`) types that are valid in the scope of a key. Their use of type-level keys adds notational and algorithmic complexity to the scoping of borrowed permissions that we avoid by using our simpler `local` permissions. On the other hand, Vault supports adoption whereby a linear permission stored in a field of a non-linear object can be treated linearly. Vault also includes annotations on methods that consume permissions similar to our change permissions. However ours are strictly more flexible because in our richer set of permissions we can specify a partial return of a permission (e.g. `unique>>immutable`).

The Cyclone language [18] has a feature similar to Vault’s keys. Cyclone also includes explicit support for borrowing through reference counting that is similar to the mechanism that underlies our `local` permissions. However, unlike our approach, it is exposed to the programmer in the syntax. Also, Cyclone allows borrowing only for permissions stored in local variables; field accesses occur via `swap`, which is awkward. In contrast, we have designed a field unpacking mechanism that supports direct field access.

Other recent work on the Plaid type system [28] integrates permissions with typestate and uses change types, providing some of the expressiveness of our system. While this other work can express the publication example from Figure 4, it has very limited

support for borrowing, and can only change field values with a swap operation, which is unnatural for programmers.

An alternative to borrowing is explicitly “threading” references from one call to another, as supported in Alms [26]. In this approach, the permission is tied to the reference; it is given up permanently when the reference is passed to a function, but the function may return the reference again along with a permission. This approach is very clear and explicit, but it is quite awkward and furthermore results in additional writes when the result reference is re-assigned to the reference variable.

Overall, the system presented in this paper is distinguished by supporting natural programming and reasoning abstractions together with a broad set of permissions including immutable, unique, and shared. As a type system defined by local rules, it is easy for programmers to follow, and separating permission flow from references makes it more succinct than systems in which references must be threaded explicitly. We hope it will serve as a robust foundation for making permission-based programming languages such as Plaid practical enough for widespread use.

5. Conclusion and Future Work

We have described a new type system for flexible borrowing of unique, shared, and immutable permissions without explicit fractions. As future work, we would like to integrate our borrowing mechanism with Plaid’s typestate features, and gain experience using the type system on larger codebases.

Acknowledgements

This material is based upon work supported by the National Science Foundation under grant #CCF-1116907, “Foundations of Permission-Based Object-Oriented Languages,” grant #CCF-0811592, “Practical Typestate Verification with Assume-Guarantee Reasoning,” and grant #1019343 to the Computing Research Association for the CIFellows Project. We thank the anonymous reviewers for their helpful feedback.

References

- [1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.
- [2] K. Bierhoff. Automated program verification made SYMLAR: SYMbolic Permissions for Lightweight Automated Reasoning. In *Onward!*, 2011.
- [3] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.
- [4] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *OOPSLA*, 2009.
- [5] B. Bokowski and J. Vitek. Confined types. In *OOPSLA*, 1999.
- [6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.
- [7] J. Boyland. Alias Burying: Unique Variables without Destructive Reads. *Software Practice and Experience*, 6(31):533–553, 2001.
- [8] J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, 2003.
- [9] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, 2001.
- [10] J. T. Boyland and W. Retert. Connecting Effects and Uniqueness With Adoption. In *POPL*, 2005.
- [11] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.
- [12] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, 2004.
- [13] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.
- [14] J.-Y. Girard. Linear logic. *Theoretical Comp. Sci.*, 50(1):1–102, 1987.
- [15] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI*, 2002.
- [16] D. Harms and B. Weide. Copying and Swapping: Influences on the design of reusable software components. *Trans. Software Engineering*, 17(5):424–435, May 1991.
- [17] S. Heule, R. Leino, P. Müller, and A. Summers. Fractional permissions without the fractions. In *ITFP*, 2011.
- [18] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *ISMM*, 2004.
- [19] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*, 1991.
- [20] R. C. Holt, P. A. Matthews, J. A. Rosselet, and J. R. Cordy. *The Turing Language: Design and Definition*. Prentice-Hall, 1988.
- [21] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, 2011.
- [22] N. H. Minsky. Towards alias-free pointers. In *ECOOP*, 1996.
- [23] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. Technical Report CMU-CS-11-142, Computer Science Department, Carnegie Mellon University, December 2011.
- [24] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*. Springer, 1998.
- [25] S. Smetsers, E. Barendsen, M. van Eekelen, and R. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Dagstuhl Seminar on Graph Transformations in Comp. Sci.*, volume 776 of *LNCS*. Springer, 1994.
- [26] J. A. Tov and R. Pucella. Practical affine types. In *POPL*, 2011.
- [27] P. Wadler. Linear types can change the world! In *Working Conf. on Programming Concepts and Methods*, 1990.
- [28] R. Wolff, R. Garcia, Éric Tanter, and J. Aldrich. Gradual typestate. In *ECOOP*, 2011.

A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language *

Adam Chlipala

University of California, Berkeley

adamc@cs.berkeley.edu

Abstract

We present a certified compiler from the simply-typed lambda calculus to assembly language. The compiler is certified in the sense that it comes with a machine-checked proof of semantics preservation, performed with the Coq proof assistant. The compiler and the terms of its several intermediate languages are given dependent types that guarantee that only well-typed programs are representable. Thus, type preservation for each compiler pass follows without any significant “proofs” of the usual kind. Semantics preservation is proved based on denotational semantics assigned to the intermediate languages. We demonstrate how working with a type-preserving compiler enables type-directed proof search to discharge large parts of our proof obligations automatically.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods, reliability; D.3.4 [*Programming Languages*]: Compilers

General Terms Languages, Verification

Keywords compiler verification, interactive proof assistants, dependent types, denotational semantics

1. Introduction

Compilers are some of the most complicated pieces of widely-used software. This fact has unfortunate consequences, since almost all of the guarantees we would like our programs to satisfy depend on the proper workings of our compilers. Therefore, proofs of compiler correctness are at the forefront of useful formal methods research, as results here stand to benefit many users. Thus, it is not surprising that recently and historically there has been much interest in this class of problems. This paper is a report on our foray into that area and the novel techniques that we developed in the process.

One interesting compiler paradigm is *type-directed compilation*, embodied in, for instance, the TIL Standard ML compiler [TMC⁺96]. Where traditional compilers use relatively type-

impoverished intermediate languages, TIL employed *typed intermediate languages* such that every intermediate program had a typing derivation witnessing its safety, up until the last few phases of compilation. This type information can be used to drive important optimizations, including *nearly tag-free garbage collection*, where the final binary comes with a table giving the type of each register or stack slot at each program point where the garbage collector may be called. As a result, there is no need to use any dynamic typing scheme for values in registers, such as tag bits or boxing.

Most of the intricacies of TIL stemmed from runtime passing of type information to support polymorphism. In the work we present here, we instead pick the modest starting point of simply-typed lambda calculus, but with the same goal: we want to compile the programs of this calculus into an idealized assembly language that uses nearly tag-free garbage collection. We will achieve this result by a series of six type-directed translations, with their typed target languages maintaining coarser and coarser grained types as we proceed. Most importantly, we prove *semantics preservation* for each translation and compose these results into a machine-checked correctness proof for the compiler. To our knowledge, there exists no other computer formalization of such a proof for a type-preserving compiler.

At this point, the reader may be dubious about just how involved a study of simply-typed lambda calculus can be. We hope that the exposition in the remainder of this paper will justify the interest of the domain. Perhaps the key difficulty in our undertaking has been effective handling of variable binding. The POPLmark Challenge [ABF⁺05] is a benchmark initiative for computer formalization of programming languages whose results have highlighted just that issue. In the many discussions it has generated, there has yet to emerge a clear winner among basic techniques for representing language constructs that bind variables in local scopes. Each of the proposed techniques leads to some significant amount of overhead not present in traditional proofs. Moreover, the initial benchmark problem doesn’t involve any component of relational reasoning, crucial for compiler correctness. We began this work as an investigation into POPLmark-style issues in the richer domain of relational reasoning.

1.1 Task Description

The source language of our compiler is the familiar simply-typed lambda calculus, whose syntax is:

$$\text{Types} \quad \tau ::= \mathbb{N} \mid \tau \rightarrow \tau$$

$$\text{Natural numbers} \quad n$$

$$\text{Variables} \quad x, y, z$$

$$\text{Terms} \quad e ::= n \mid x \mid e e \mid \lambda x : \tau. e$$

Application, the third production for e , associates to the left, as usual. We will elaborate shortly on the language’s semantics.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI’07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006...\$5.00.

Our target language is an idealized assembly language, with infinitely many registers and memory cells, each of which holds unbounded natural numbers. We model interaction with a runtime system through special instructions. The syntax of the target language is:

$$\begin{array}{lll} \text{Registers} & r \\ \text{Operands} & o ::= r \mid n \mid \text{new } \langle \vec{r}, \vec{r} \rangle \mid \text{read } \langle r, n \rangle \\ \text{Instructions} & i ::= r := o; i \mid \text{jump } r \\ \text{Programs} & p ::= \langle \vec{i}, i \rangle \end{array}$$

As instruction operands, we have registers, constant naturals, and allocation and reading of heap-allocated records. The new operand takes two arguments: a list of root registers and a list of registers holding the field values for the object to allocate. The semantics of the target language are such that new is allowed to perform garbage collections at will, and the root list is the usual parameter required for sound garbage collection. In fact, we will let new rearrange memory however it pleases, as long as the result is indistinguishable from having simply allocated a new record, from the point of view of the specified roots. The read instruction determines which runtime system-specific procedure to use to read a given field of a record.

A program is a list of basic blocks plus an initial basic block, where execution begins. A basic block is a sequence of register assignments followed by an indirect jump. The basic blocks are indexed in order by the natural numbers for the purposes of these jumps. We will additionally consider that a jump to the value 0 indicates that the program should halt, and we distinguish one register that is said to hold the program's result at such points.

We are now ready to state informally the theorem whose proof is the goal of this work:

THEOREM 1 (Informal statement of compiler correctness). *Given a term e of the simply-typed lambda calculus of type N , the compilation of e is an assembly program that, when run, terminates with the same result as we obtain by running e .*

Our compiler itself is implemented entirely within the Coq proof assistant [BC04]. Coq's logic doubles as a functional programming language. Through the *program extraction* process, the more exotic features of a development in this logic can be erased in a semantics-preserving way, leaving an OCaml program that can be compiled to efficient object code. In this way, we obtain a traditional executable version of our compiler. We ignore issues of parsing in this work, so our compiler must be composed with a parser. Assuming a fictitious machine that runs our idealized assembly language, the only remaining piece to be added is a pretty-printer from our abstract syntax tree representation of assembly programs to whatever format that machine requires.

1.2 Contributions

We summarize the contributions of this work as follows:

- It includes what is to our knowledge the first total correctness proof of an entire type-preserving compiler.
- It gives the proof using denotational semantics, in contrast to the typical use of operational semantics in related work.
- The whole formal development is carried out in a completely rigorous way with the Coq proof assistant [BC04], yielding a proof that can be checked by a machine using a relatively small trusted code base. Our approach is based on a general methodology for representing variable binding and denotation functions in Coq.
- We sketch a generic programming system that we have developed for automating construction of syntactic helper functions

over de Bruijn terms [dB72], as well as generic correctness proofs about these functions. In addition, we present the catalogue of generic functions that we found sufficient for this work.

- Finally, we add to the list of pleasant consequences of making your compiler type-directed or type-preserving. In particular, we show how dependently-typed formalizations of type-preserving compilers admit particularly effective automated proof methods, driven by type information.

1.3 Outline

In the next section, we present our compiler's intermediate languages and elaborate on the semantics of the source and target languages. Following that, we run through the basic elements of implementing a compiler pass, noting challenges that arise in the process. Each of the next sections addresses one of these challenges and our solution to it. We discuss how to mechanize typed programming languages and transformations over them, how to use generic programming to simplify programming with dependently-typed abstract syntax trees, and how to apply automated theorem proving to broad classes of proof obligations arising in certification of type-preserving compilers. After this broad discussion, we spend some time discussing interesting features of particular parts of our formalization. Finally, we provide some statistics on our implementation, discuss related work, and conclude.

2. The Languages

In this section, we present our source, intermediate, and target languages, along with their static and dynamic semantics. Our language progression is reminiscent of that from the paper “From System F to Typed Assembly Language” [MWCG99]. The main differences stem from the fact that we are interested in meaning preservation, not just type safety. This distinction makes our task both harder and easier. Naturally, semantics preservation proofs are more difficult than type preservation proofs. However, the fact that we construct such a detailed understanding of program behavior allows us to retain less type information in later stages of the compiler.

The mechanics of formalizing these languages in Coq is the subject of later sections. We will stick to a “pencil and paper formalization” level of detail in this section. We try to use standard notations wherever possible. The reader can rest assured that anything that seems ambiguous in the presentation here is clarified in the mechanization.

2.1 Source

The syntax of our source language (called “Source” hereafter) was already given in Section 1.1. The type system is the standard one for simply-typed lambda calculus, with judgments of the form $\Gamma \vdash e : \tau$ which mean that, with respect to the type assignment to free variables provided by Γ , term e has type τ .

Following usual conventions, we require that, for any typing judgment that may be stated, let alone verified to hold, all free and bound variables are distinct, and all free variables of the term appear in the context. In the implementation, the first condition is discharged by using de Bruijn indices, and the second condition is covered by the dependent typing rules we will assign to terms.

Next we need to give Source a dynamic semantics. For this and all our other languages, we opted to use denotational semantics. The utility of this choice for compiler correctness proofs has been understood for a while, as reifying a program phrase's meaning as a mathematical object makes it easy to transplant that meaning to new contexts in modeling code transformations.

Our semantics for Source is:

$$\begin{aligned}
\llbracket \tau \rrbracket & : \text{types} \rightarrow \text{sets} \\
\llbracket \mathbb{N} \rrbracket & = \mathbb{N} \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket & = \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket \\
\\
\llbracket \Gamma \rrbracket & : \text{contexts} \rightarrow \text{sets} \\
\llbracket \mathbb{I} \rrbracket & = \text{unit} \\
\llbracket \Gamma, x : \tau \rrbracket & = \llbracket \Gamma \rrbracket \times \llbracket \tau \rrbracket \\
\\
\llbracket e \rrbracket & : [\Gamma \vdash e : \tau] \rightarrow \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket \\
\llbracket n \rrbracket \sigma & = \bar{n} \\
\llbracket x \rrbracket \sigma & = \sigma(x) \\
\llbracket e_1 e_2 \rrbracket \sigma & = \llbracket e_1 \rrbracket \sigma \llbracket e_2 \rrbracket \sigma \\
\llbracket \lambda x : \tau. e \rrbracket \sigma & = \lambda x : \llbracket \tau \rrbracket. \llbracket e \rrbracket(\sigma, x)
\end{aligned}$$

The type of the term denotation function (used in expressions of the form $\llbracket e \rrbracket$) indicates that it is a function whose domain is *typing derivations* for terms and whose range depends on the particular Γ and τ that appear in that derivation. As a result, we define denotations only for well-typed terms.

Every syntactic class is being compiled into a single meta language, Coq’s Calculus of Inductive Constructions (CIC) [BC04]. However, we will not expect any special knowledge of this formal system from the reader. Various other type theories could be substituted for CIC, and standard set theory would do just as well for this informal presentation.

We first give each Source type a meaning by a recursive translation into sets. Note that, throughout this paper, we overload constructs like the function type constructor \rightarrow that are found in both our object languages and the meta language CIC, in an effort to save the reader from a deluge of different arrows. The variety of arrow in question should always be clear from context. With this convention in mind, the type translation for Source is entirely unsurprising. \mathbb{N} denotes the mathematical set of natural numbers, as usual.

We give a particular type theoretical interpretation of typing contexts as tuples, and we then interpret a term that has type τ in context Γ as a function from the denotation of Γ into the denotation of τ . The translation here is again entirely standard. For the sake of conciseness, we allow ourselves to be a little sloppy with notations like $\sigma(x)$, which denotes the proper projection from the tuple σ , corresponding to the position x occupies in Γ . We note that we use the *meta language’s lambda binder* to encode the object language’s lambda binder in a natural way, in an example closely related to higher-order abstract syntax [PE88].

2.2 Linear

Our first compilation step is to convert Source programs into a form that makes *execution order* explicit. This kind of translation is associated with continuation-passing style (CPS), and the composition of the first two translations accomplishes a transformation to CPS. The result of the first translation is the Linear language, and we now give its syntax. It inherits the type language of Source, though we interpret the types differently.

$$\begin{aligned}
\text{Operands } o & ::= n \mid x \mid \lambda x : \tau. e \\
\text{Terms } e & ::= \text{let } x = o \text{ in } e \mid \text{throw } \langle o \rangle \mid x \ y \ z
\end{aligned}$$

Linear terms are linearized in the sense that they are sequences of binders of primitive operands to variables, followed by either a “throw to the current continuation” or a function call. The function call form shows that functions take two arguments: first, the normal

argument; and second, a function to call with the result. The function and both its arguments must be variables, perhaps bound with earlier lets.

For reasons of space, we omit the standard typing rules for Linear and proceed to its denotational semantics. Recall, however, that the denotation functions are only defined over well-typed terms.

$$\begin{aligned}
\llbracket \mathbb{H} \rrbracket & : \text{types} \rightarrow \text{sets} \\
\llbracket \mathbb{N} \rrbracket & = \mathbb{N} \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket & = \llbracket \tau_1 \rrbracket \rightarrow (\llbracket \tau_2 \rrbracket \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\
\\
\llbracket o \rrbracket & : [\Gamma \vdash o : \tau] \rightarrow \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket \\
\llbracket n \rrbracket \sigma & = \bar{n} \\
\llbracket x \rrbracket \sigma & = \sigma(x) \\
\llbracket \lambda x : \tau. e \rrbracket \sigma & = \lambda x : \llbracket \tau \rrbracket. \llbracket e \rrbracket(\sigma, x) \\
\\
\llbracket e \rrbracket & : [\Gamma \vdash o : \tau] \rightarrow \llbracket \Gamma \rrbracket \rightarrow (\llbracket \mathbb{H} \rrbracket \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\
\llbracket \text{let } x = o \text{ in } e \rrbracket \sigma & = \llbracket e \rrbracket(\sigma, \llbracket o \rrbracket \sigma) \\
\llbracket \text{throw } \langle o \rangle \rrbracket \sigma & = \lambda k. k(\llbracket o \rrbracket \sigma) \\
\llbracket x \ y \ z \rrbracket \sigma & = \lambda k. \sigma(x)(\sigma(y))(\lambda v. k(\sigma(z)(v)))
\end{aligned}$$

We choose the natural numbers as the result type of programs. Functions are interpreted as accepting continuations that return naturals when given a value of the range type. The let case relies on the implicit fact that the newly-bound variable x falls at the end of the proper typing context for the body e . The most interesting case is the last one listed, that for function calls. We call the provided function with a new continuation created by composing the current continuation with the continuation supplied through the variable z .

We use Linear as our first intermediate language instead of going directly to standard CPS because the presence of distinguished throw terms makes it easier to optimize term representation by splicing terms together. This separation is related to the issue of “administrative redexes” in standard two-phase CPS transforms [Plo75].

2.3 CPS

The next transformation finishes the job of translating Linear into genuine CPS form, and we call this next target language CPS.

$$\begin{aligned}
\text{Types } \tau & ::= \mathbb{N} \mid \vec{\tau} \rightarrow \mathbb{N} \\
\text{Operands } o & ::= n \mid x \mid \lambda \vec{x} : \vec{\tau}. e \\
\text{Terms } e & ::= \text{let } x = o \text{ in } e \mid x \vec{y}
\end{aligned}$$

Compared to Linear, the main differences we see are that functions may now take multiple arguments and that we have collapsed throw and function call into a single construct. In our intended use of CPS, functions will either correspond to source-level functions and take two arguments, or they will correspond to continuations and take single arguments. Our type language has changed to reflect that functions no longer return, but rather they lead to final natural number results.

We omit discussion of the semantics of CPS, since the changes from Linear are quite incremental.

2.4 CC

The next thing the compiler does is closure convert CPS programs, hoisting all function definitions to the top level and changing those functions to take records of their free variables as additional arguments. We call the result language CC, and here is its syntax.

$$\begin{aligned}
\text{Types } \tau & ::= \mathbb{N} \mid \vec{\tau} \rightarrow \mathbb{N} \mid \bigotimes \vec{\tau} \mid \vec{\tau} \times \vec{\tau} \rightarrow \mathbb{N} \\
\text{Operands } o & ::= n \mid x \mid \langle x, \vec{y} \rangle \mid \pi_i x \\
\text{Terms } e & ::= \text{let } x = o \text{ in } e \mid x \vec{y} \\
\text{Programs } p & ::= \text{let } x = (\lambda \vec{y} : \vec{\tau}. e) \text{ in } p \mid e
\end{aligned}$$

We have two new type constructors: $\otimes \vec{\tau}$ is the multiple-argument product type of records whose fields have the types given by $\vec{\tau}$. $\vec{\tau} \times \vec{\tau} \rightarrow \mathbb{N}$ is the type of *code pointers*, specifying first the type of the closure environment expected and second the types of the regular arguments.

Among the operands, we no longer have an anonymous function form. Instead, we have $\langle x, \vec{y} \rangle$, which indicates the creation of a closure with code pointer x and environment elements \vec{y} . These environment elements are packaged atomically into a record, and the receiving function accesses the record's elements with projection operand form $\pi_i x$. This operand denotes the i th element of record x .

While we have added a number of features, there are no surprises encountered in adapting the previous semantics, so we will proceed to the next language.

2.5 Alloc

The next step in compilation is to make allocation of products and closures explicit. Since giving denotational semantics to higher-order imperative programs is tricky, we decided to perform “code flattening” as part of the same transformation. That is, we move to first-order programs with fixed sets of numbered code blocks, and every function call refers to one of these blocks by number. Here is the syntax of this language, called Alloc.

$$\begin{array}{lll} \text{Types} & \tau ::= & \mathbb{N} \mid \text{ref} \mid \vec{\tau} \rightarrow \mathbb{N} \\ \text{Operands} & o ::= & n \mid x \mid \langle n \rangle \mid \text{new } \langle \vec{x} \rangle \mid \pi_i x \\ \text{Terms} & e ::= & \text{let } x = o \text{ in } e \mid x \vec{y} \\ \text{Programs} & p ::= & \text{let } (\lambda \vec{y} : \vec{\tau}. e) \text{ in } p \mid e \end{array}$$

The first thing to note is that this phase is the first in which we lose type information: we have a single type ref for all heap references, forgetting what we know about record field types. To compensate for this loss of information, we will give Alloc programs a semantics that allows them to fail when they make “incorrect guesses” about the types of heap cells.

Our set of operands now includes both natural number constants n and code pointer constants $\langle n \rangle$. We also have a generic record allocation construct in place of the old closure constructor, and we retain projections from records.

This language is the first to take a significant departure from its predecessors so far as dynamic semantics is concerned. The key difference is that Alloc admits non-terminating programs. While variable scoping prevented cycles of function calls in CC and earlier, we lose that restriction with the move to a first-order form. This kind of transformation is inevitable at some point, since our target assembly language has the same property.

Domain theory provides one answer to the questions that arise in modeling non-terminating programs denotationally, but it is difficult to use the classical work on domain theory in the setting of constructive type theory. One very effective alternative comes in the form of the *co-inductive types* [Gim95] that Coq supports. A general knowledge of this class of types is not needed to understand what follows. We will confine our attention to one co-inductive type, a sort of *possibly-infinite streams*. We define this type with the infinite closure of this grammar:

$$\text{Traces } T ::= n \mid \perp \mid \star, T$$

In other words, a *trace* is either an infinite sequence of stars or a finite sequence of stars followed by a natural number or a bottom value. The first of these possibilities will be the denotation of a non-terminating program, the second will denote a program returning an answer, and the third will denote a program that “crashes.”

Now we are ready to start modeling the semantics of Alloc. First, we need to fix a representation of the heap. We chose an abstract representation that identifies heaps with lists of lists of

tagged fields, standing for the set of finite-size records currently allocated. Each field consists of a tag, telling whether or not it is a pointer; and a data component. As we move to lower languages, these abstract heaps will be mapped into more conventional flat, untyped heaps. We now define some domains to represent heaps, along with a function for determining the proper tag for a type:

$$\begin{array}{ll} \mathbb{C} & = \{ \text{Traced}, \text{Untraced} \} \times \mathbb{N} \\ \mathbb{M} & = \text{list } (\text{list } \mathbb{C}) \\ \text{tagof(N)} & = \text{Untraced} \\ \text{tagof(ref)} & = \text{Traced} \\ \text{tagof}(\vec{\tau} \rightarrow \mathbb{N}) & = \text{Untraced} \end{array}$$

Next we give a semantics to operands. We make two different choices here than we did before. First, we allow execution of operands to *fail* when a projection reads a value from the heap and finds that it has the wrong tag. Second, the denotations of operands must be heap transformers, taking the heap as an extra argument and returning a new one to reflect any changes. We also set the convention that program counter 0 denotes the distinguished top-level continuation of the program that, when called, halts the program with its first argument as the result. Any other program counter $n + 1$ denotes the n th function defined in the program.

$$\begin{array}{ll} \llbracket o \rrbracket & : [\Gamma \vdash o : \tau] \rightarrow \llbracket \Gamma \rrbracket \rightarrow \mathbb{M} \rightarrow (\mathbb{M} \times \mathbb{N}) \cup \{ \perp \} \\ \llbracket n \rrbracket \sigma m & = (m, \bar{n}) \\ \llbracket x \rrbracket \sigma m & = (m, \sigma(x)) \\ \llbracket \langle n \rangle \rrbracket \sigma m & = (m, \bar{n+1}) \\ \llbracket \text{new } \langle \vec{x} \rangle \rrbracket \sigma m & = (m \oplus [\sigma(\vec{x})], |m|) \\ \llbracket \pi_i x \rrbracket \sigma m & = \text{if } m_{\sigma(x), i} = (\text{tagof}(\tau), v), \text{then: } (m, v) \\ & \text{else: } \perp \end{array}$$

We write \oplus to indicate list concatenation, and the notation $\sigma(\vec{x})$ to denote looking up in σ the value of each variable in \vec{x} , forming a list of results where each is tagged appropriately based on the type of the source variable. The new case returns the length of the heap because we represent heap record addresses with their zero-based positions in the heap list.

Terms are more complicated. While one might think of terms as potentially non-terminating, we take a different approach here. The denotation of every term is a terminating program that returns *the next function call that should be made*, or signals an error with a \perp value. More precisely, a successful term execution returns a tuple of a new heap, the program counter of the function to call, and a list of natural numbers that are to be the actual arguments.

$$\begin{array}{ll} \llbracket e \rrbracket & : [\Gamma \vdash e : \tau] \rightarrow \llbracket \Gamma \rrbracket \rightarrow \mathbb{M} \\ & \quad \rightarrow (\mathbb{M} \times \mathbb{N} \times \text{list } \mathbb{N}) \cup \{ \perp \} \\ \llbracket \text{let } x = o \text{ in } e \rrbracket \sigma m & = \text{if } \llbracket o \rrbracket \sigma m = (m', v) \text{ then: } \llbracket e \rrbracket (\sigma, v) m' \\ & \quad \text{else: } \perp \\ \llbracket x \vec{y} \rrbracket \sigma m & = (m, \sigma(x), \sigma(\vec{y})) \end{array}$$

Finally, we come to the programs, where we put our trace domain to use. Since we have already converted to CPS, there is no need to consider any aspect of a program’s behavior but its result. Therefore, we interpret programs as functions from heaps to traces. We write :: for the binary operator that concatenates a new element to the head of a list, and we write \vec{x}_{pc} for the lookup operation that extracts from the function definition list \vec{x} the pc th element.

$$\begin{aligned}
[p] & : [\Gamma \vdash \text{let } \vec{x} \text{ in } e] \rightarrow [\Gamma] \rightarrow \mathbb{M} \rightarrow \text{Trace} \\
\llbracket \text{let } \vec{x} \text{ in } e \rrbracket \sigma m & = \text{if } \llbracket e \rrbracket \sigma m = (m', 0, v :: \vec{n}) \text{ then: } v \\
& \quad \text{else if } \llbracket e \rrbracket \sigma m = (m', pc + 1, \vec{n}) \text{ then:} \\
& \quad \quad \text{if } \vec{x}_{pc} = \lambda \vec{y} : \vec{\tau}. e' \text{ and } |\vec{y}| = |\vec{n}| \text{ then:} \\
& \quad \quad \quad \star, \llbracket \text{let } \vec{x} \text{ in } e' \rrbracket \vec{n} m \\
& \quad \quad \text{else: } \perp \\
& \quad \text{else: } \perp
\end{aligned}$$

Though we have not provided details here, Coq's co-inductive types come with some quite stringent restrictions, designed to prevent unsound interactions with proofs. Our definition of program denotation is designed to satisfy those restrictions. The main idea here is that functions defined as *co-fixed points* must be “sufficiently productive”; for our trace example, a co-recursive call may only be made after at least one token has already been added to the stream in the current call. This restriction is the reason for including the seemingly information-free stars in our definition of traces. As we have succeeded in drafting a definition that satisfies this requirement, we are rewarded with a function that is not just an arbitrary relational specification, but rather a *program* that Coq is able to execute, resulting in a Haskell-style lazy list.

2.6 Flat

We are now almost ready to move to assembly language. Our last stop before then is the Flat language, which differs from assembly only in maintaining the abstract view of the heap as a list of tagged records. We do away with variables and move instead to an infinite bank of registers. Function signatures are expressed as maps from natural numbers to types, and Flat programs are responsible for shifting registers around to compensate for the removal of the built-in stack discipline that variable binding provided.

Types	τ	$::=$	$\mathbb{N} \mid \text{ref} \mid \Delta \rightarrow \mathbb{N}$
Typings	Δ	$=$	$\mathbb{N} \rightarrow \tau$
Registers	r		
Operands	o	$::=$	$n \mid r \mid \langle n \rangle \mid \text{new } \langle \vec{r} \rangle \mid \pi_i r$
Terms	e	$::=$	$r := o; e \mid \text{jump } r$
Programs	p	$::=$	$\text{let } e \text{ in } p \mid e$

In the style of Typed Assembly Language, to each static point in a Flat program we associate a typing Δ expressing our expectations of register types whenever that point is reached dynamically. It is crucial that we keep this typing information, since we will use it to create garbage collector root tables in the next and final transformation.

Since there is no need to encode any variable binding for Flat, its denotational semantics is entirely routine. The only exception is that we re-use the trace semantics from Alloc.

2.7 Asm

Our idealized assembly language Asm was already introduced in Section 1.1. We have by now already provided the main ingredients needed to give it a denotational semantics. We re-use the trace-based approach from the last two languages. The difference we must account for is the shift from abstract to concrete heaps, as well as the fact that Asm is parametric in a runtime system.

We should make it clear that we have not verified any runtime system or garbage collector, but only stated conditions that they ought to satisfy and proved that those conditions imply the correctness of our compiler. Recent work on formal certification of garbage collectors [MSLL07] gives us hope that the task we have omitted is not insurmountable.

In more detail, our formalization of Asm starts with the definition of the domain \mathbb{H} of concrete heaps:

$$\mathbb{H} = \mathbb{N} \rightarrow \mathbb{N}$$

A runtime system provides new and read operations. The read operation is simpler:

$$\text{read} : \mathbb{H} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

For a heap, a pointer to a heap-allocated record, and a constant field offset within that record, read should return that field's current value. Runtime systems may make different decisions on concrete layout of records. For instance, there are several ways of including information on which fields are pointers. Note that read is an arbitrary Coq function, not a sequence of assembly instructions, which should let us reason about many different runtime system design decisions within our framework.

The new operation is more complicated, as it is designed to facilitate garbage collector operation.

$$\begin{aligned}
\text{new} : & \mathbb{H} \times \text{list } \mathbb{N} \times \text{list } \mathbb{N} \\
& \rightarrow \mathbb{H} \times \text{list } \mathbb{N} \times \mathbb{N}
\end{aligned}$$

Its arguments are the current heap, the values to use to initialize the fields of the record being allocated, and the values of all live registers holding pointer values. The idea is that, in the course of fulfilling the allocation request, the runtime system may rearrange memory however it likes, so long as things afterward look the same as before to any type-safe program, from the perspective of the live registers. A return value of new gives the modified heap, a fresh set of values for the pointer-holding registers (i.e., the garbage collection roots, which may have been moved by a copying collector), and a pointer to the new record in the new heap.

To state the logical conditions imposed on runtime systems, we will need to make a definition based on the abstract heap model of earlier intermediate languages:

DEFINITION 1 (Pointer isomorphism). *We say that pointers $p_1, p_2 \in \mathbb{N}$ are isomorphic with respect to abstract heaps $m_1, m_2 \in \mathbb{M}$ iff:*

1. *The p_1 th record of m_1 and p_2 th record of m_2 have the same number of fields.*
2. *If the i th field of the p_1 th record of m_1 is tagged Untraced, then the i th field of the p_2 th record of m_2 is also tagged Untraced, and the two fields have the same value.*
3. *If the i th field of the p_1 th record of m_1 is tagged Traced, then the i th field of the p_2 th record of m_2 is also tagged Traced, and the two fields contain isomorphic pointers.*

The actual definition is slightly more involved but avoids this unqualified self-reference.

To facilitate a parametric translation soundness proof, a candidate runtime system is required to provide a concretization function:

$$\gamma : \mathbb{M} \rightarrow \mathbb{H}$$

For an abstract heap m , $\gamma(m)$ is the concrete heap with which the runtime system's representation conventions associate it. We also overload γ to stand for a different function for concretizing pointers. We abuse notation by applying γ to various different kinds of objects, where it's clear how they ought to be converted from abstract to concrete in terms of the two primary γ translations; and, while some of these γ functions really need to take the abstract heap as an additional argument, we omit it for brevity where the proper value is clear from context. The runtime system must come with proofs that its new and read operations satisfy the appropriate commutative diagrams with these concretization functions.

To give a specific example, we show the more complicated condition between the two operations, that for new.

THEOREM 2 (Correctness of a new implementation). *For any abstract heap m , tagged record field values \vec{v} (each in \mathbb{C}), and register root set values \vec{r} 's, there exist new abstract heap m' , new root values $\vec{r}'s$, and new record address a such that:*

1. $\text{new}(\gamma(m), \gamma(\vec{v}), \gamma(\vec{r})) = (\gamma(m'), \gamma(\vec{r}'), \gamma(a))$
2. For every pair of values p and p' in \vec{r} 's and $\vec{r}'s$, respectively, p and p' are isomorphic with respect to $m \oplus [\vec{v}]$ and m' .
3. $|m|$ and a are isomorphic with respect to $m \oplus [\vec{v}]$ and m' .

To understand the details of the last two conditions, recall that, in the abstract memory model, we allocate new records at the end of the heap, which is itself a list of records. The length of the heap before an allocation gives the proper address for the next record to be allocated.

3. Implementation Strategy Overview

Now that we have sketched the basic structure of our compiler, we move to introducing the ideas behind its implementation in the Coq proof assistant. In this section, we single out the first compiler phase and walk through its implementation and proof at a high level, noting the fundamental challenges that we encounter along the way. We summarize our solution to each challenge and then discuss each in more detail in a later section.

Recall that our first phase is analogous to the first phase of a two-phase CPS transform. We want to translate the Source language to the Linear language.

Of course, before we can begin writing the translation, we need to represent our languages! Thus, our first challenge is to choose a representation of each language using Coq types. Our solution here is based on de Bruijn indices and dependently-typed abstract syntax trees. Our use of dependent typing will ensure that representable terms are free of dangling variable references by encoding a term's free variable set in its type. Moreover, we will *combine typing derivations and terms*, essentially representing each term as its typing derivation. In this way, only *well-typed* terms are representable.

Now we can begin writing the translation from Source to Linear. We have some choices about how to represent translations in Coq, but, with our previous language representation choice, it is quite natural to represent translations as dependently-typed Coq functions. Coq's type system will ensure that, when a translation is fed a well-typed input program, it produces a well-typed output program. We can use Coq's *program extraction* facility to produce OCaml versions of our translations by erasing their dependently-typed parts.

This strategy is very appealing, and it is the one we have chosen, but it is not without its inconveniences. Since we represent terms as their typing derivations, standard operations like bringing a new, unused variable into scope are not no-ops like they are with some other binding representations. These variable operations turn out to correspond to standard theorems about typing contexts. For instance, bringing an unused variable into scope corresponds to a *weakening* lemma. These syntactic functions are tedious to write for each new language, especially when dealing with strong dependent types. Moreover, their implementations have little to do with details of particular languages. As a result, we have been able to create a generic programming system that produces them automatically for arbitrary languages satisfying certain criteria. We not only produce the functions automatically, but we also produce proofs that they commute with arbitrary compositional denotation functions in the appropriate, function-specific senses.

Now assume that we have our translation implemented. Its type ensures that it preserves well-typedness, but we also want to prove that it preserves meaning. What proof technique should we use? The technique of logical relations [Plo73] is the standard for this sort of task. We characterize the relationships between program entities and their compilations using relations defined recursively on type structure. Usual logical relations techniques for denotational semantics translate very effectively into our setting, and we are able to take good advantage of the expressiveness of our meta language CIC in enabling succinct definitions.

With these relations in hand, we reach the point where most pencil-and-paper proofs would say that the rest follows by “routine inductions” of appropriate kinds. Unfortunately, since we want to convince the Coq proof checker, we will need to provide considerably more detail. There is no magic bullet for automating proofs of this sophistication, but we did identify some techniques that worked surprisingly well for simplifying the proof burden. In particular, since our compiler preserves type information, we were able to automate significant portions of our proofs using type-based heuristics. The key insight was the possibility of using *greedy quantifier instantiation*, since the dependent types we use are so particular that most logical quantifiers have domains compatible with just a single subterm of a proof sequent.

4. Representing Typed Languages

We begin our example by representing the target language of its transformation. The first step is easy; we give a standard algebraic datatype definition of the type language shared by Source and Linear.

```
type   : set
Nat    : type
Arrow  : type → type → type
```

Things get more interesting when we go to define our term languages. We want to use dependent types to ensure that only terms with object language typing derivations are representable, so we make our classes of operands and terms indexed type families whose indices tell us their object language types. Coq's expressive dependent type system admits simple implementations of this idea. Its *inductive types* are a generalization of the generalized algebraic datatypes [She04] that have become popular recently.

We represent variables with de Bruijn indices. In other words, a variable is represented as a natural number counting how many binders outward in lexical scoping order one must search to find its binder. Since we are using dependent typing, variables are more than just natural numbers. We represent our de Bruijn contexts as lists of types, and variables are in effect constructive proofs that a particular type is found in a particular context. A generic type family `var` encapsulates this functionality.

We define operands and terms as two mutually-inductive Coq types. Below, the standard notation Π is used for a dependent function type. The notation $\Pi x : \tau_1, \tau_2$ denotes a function from τ_1 to τ_2 , where the variable x is *bound* in the range type τ_2 , indicating that the function's result type may depend on the *value* of its argument. We elide type annotations from Π types where they are clear from context.

```
Iprimop  : list type → type → set
LConst   : ΠΓ, N → Iprimop Γ Nat
LVar     : ΠΓ, Πτ, var Γ τ → Iprimop Γ τ
LLam    : ΠΓ, Πτ₁, Πτ₂, lterm (τ₁ :: Γ) τ₂
          → Iprimop Γ (Arrow τ₁ τ₂)
```

```

Inductive sty : Set :=
| SNat : sty
| SArrow : sty -> sty -> sty.

Inductive sterm : list sty -> sty -> Set :=
| SVar : forall G t,
  Var G t
  -> sterm G t
| SLam : forall G dom ran,
  sterm (dom :: G) ran
  -> sterm G (SArrow dom ran)
| SApp : forall G dom ran,
  sterm G (SArrow dom ran)
  -> sterm G dom
  -> sterm G ran
| SConst : forall G, nat -> sterm G SNat.

Fixpoint styDenote (t : sty) : Set :=
match t with
| SNat => nat
| SArrow t1 t2 => styDenote t1 -> styDenote t2
end.

Fixpoint stermDenote (G : list sty) (t : sty)
  (e : sterm G t) {struct e}
: Subst styDenote G -> styDenote t :=
match e in (sterm G t)
  return (Subst styDenote G -> styDenote t) with
| SVar _ _ v => fun s =>
  VarDenote v s
| SLam _ _ _ e' => fun s =>
  fun x => stermDenote e' (SCons x s)
| SApp _ _ _ e1 e2 => fun s =>
  (stermDenote e1 s) (stermDenote e2 s)
| SConst _ n => fun _ => n
end.

```

Figure 1. Coq source code of Source syntax and semantics

```

lterm   : list type → type → set
LLet    : ΠΓ, Πτ1, Πτ2, lprimop Γ τ1
          → lterm (τ1 :: Γ) τ2 → lterm Γ τ2
LThrow  : ΠΓ, Πτ, lprimop Γ τ → lterm Γ τ
LApp    : ΠΓ, Πτ1, Πτ2, Πτ3, var Γ (Arrow τ1 τ2)
          → var Γ τ1 → var Γ (Arrow τ2 τ3)
          → lterm Γ τ3

```

In Coq, all constants are defined in the same syntactic class, as opposed to using separate type and term languages. Thus, for instance, lprimop and LConst are defined “at the same level,” though the type of the former tells us that it is “a type.”

As an example of the encoding, the identity term $\lambda x : \mathbb{N}.\ \text{throw } \langle x \rangle$ is represented as:

`LLam [] Nat Nat (LThrow [Nat] Nat (LVar [Nat] Nat First))`

where First is a constructor of var indicating the lexically innermost variable.

To give an idea of the make-up of our real implementation, we will also provide some concrete Coq code snippets throughout this article. The syntax and static and dynamic semantics of our source language are simple enough that we can show them here in their

entirety in Figure 1. Though Coq syntax may seem strange at first to many readers, the structure of these definitions really mirrors their informal counterparts exactly. These snippets are only meant to give a flavor of the project. We describe in Section 10 how to obtain the complete project source code, for those who want a more in-depth treatment.

5. Representing Transformations

We are able to write the linearization transformation quite naturally using our de Bruijn terms. We start this section with a less formal presentation of it.

We first define an auxiliary operation $e_1 \overset{u}{\bullet} e_2$ that, given two linear terms e_1 and e_2 and a variable u free in e_2 , returns a new linear term equivalent to running e_1 and throwing its result to e_2 by binding that result to u .

$$\begin{aligned} (\text{let } y = o \text{ in } e_1) \overset{u}{\bullet} e_2 &= \text{let } y = o \text{ in } (e_1 \overset{u}{\bullet} e_2) \\ (\text{throw } \langle o \rangle) \overset{u}{\bullet} e &= \text{let } u = o \text{ in } e \\ (x \ y \ z) \overset{u}{\bullet} e &= \text{let } f = (\lambda v. \text{let } g = (\lambda u. e) \text{ in } z \ v \ g) \\ &\quad \text{in } x \ y \ f \end{aligned}$$

Now we can give the linearization translation itself.

$$\begin{aligned} \lfloor n \rfloor &= \text{throw } \langle n \rangle \\ \lfloor x \rfloor &= \text{throw } \langle x \rangle \\ \lfloor e_1 \ e_2 \rfloor &= \lfloor e_1 \rfloor \overset{u}{\bullet} (\lfloor e_2 \rfloor \overset{v}{\bullet} (\text{let } f = (\lambda x. \text{throw } \langle x \rangle) \text{ in } u \ v \ f)) \\ \lfloor \lambda x : \tau. \ e \rfloor &= \text{throw } \langle \lambda x : \tau. \lfloor e \rfloor \rangle \end{aligned}$$

This translation can be converted rather directly into Coq recursive function definitions. The catch comes as a result of our strong dependent types for program syntax. The Coq type checker is not able to verify the type-correctness of some of the clauses above.

For a simple example, let us focus on part of the linearization case for applications. There we produce terms of the form $\lfloor e_1 \rfloor \overset{u}{\bullet} (\lfloor e_2 \rfloor \overset{v}{\bullet} \dots)$. The term e_2 originally occurred in some context Γ . However, here $\lfloor e_2 \rfloor$, the compilation of e_2 , is used in a context formed by adding an additional variable u to Γ . We know that this transplantation is harmless and ought to be allowed, but the Coq type checker flags this section as a type error.

We need to perform an explicit coercion that adjusts the de Bruijn indices in $\lfloor e_2 \rfloor$. The operation we want corresponds to a weakening lemma for our typing judgment: “If $\Gamma \vdash e : \tau$, then $\Gamma, x : \tau' \vdash e : \tau$ when x is not free in e .” Translating this description into our formalization with inductive types from the last section, we want a function:

`weakenFront : ΠΓ, Πτ, lterm Γ τ → Πτ', lterm (τ' :: Γ) τ`

It is possible to write a custom weakening function for linearized terms, but nothing about the implementation is specific to our language. There is a generic recipe for building weakening functions, based only on an understanding of where variable binders occur. To keep our translations free of such details, we have opted to create a generic programming system that builds these syntactic helper functions for us. It is the subject of the next section.

When we insert these coercions where needed, we have a direct translation of our informal compiler definition into Coq code. Assuming for now that the coercion functions have already been generated, we can give as an example of a real implementation the Coq code for the main CPS translation, in Figure 2. We lack the space to describe the code in detail, but we point out that the `Next` and `First` constructors are used to build de Bruijn variables in unary form. `compose` is the name of the function corresponding to our informal $\overset{u}{\bullet}$ operator. `Lterm` is a module of helper functions gen-

```

Fixpoint cps (G : list sty) (t : sty)
  (e : sterm G t) {struct e}
  : lterm G t :=
match e in (sterm G t) return (lterm G t) with
| SVar _ _ v => LThrow (LVar v)
| SConst _ n => LThrow (LConst _ n)
| SLam _ _ _ e' => LThrow (LLam (cps e'))
| SApp _ _ _ e1 e2 =>
  compose (cps e1)
  (compose (Lterm.weakenFront _ (cps e2))
    (LBind
      (LLam (LThrow (LVar First)))
      (LApply
        (Next (Next First))
        (Next First)
        First)))
end.

```

Figure 2. Coq source code of the main CPS translation

erated automatically, and it includes the coercion `weakenFront` described earlier. Let us, then, turn to a discussion of the generic programming system that produces it.

6. Generic Syntactic Functions

A variety of these syntactic helper functions come up again and again in formalization of programming languages. We have identified a few primitive functions that seemed to need per-language implementations, along with a number of derived functions that can be implemented parametrically in the primitives. Our generic programming system writes the primitive functions for the user and then automatically instantiates the parametric derived functions. We present here a catalogue of the functions that we found to be important in this present work. All operate over some type family term parameterized by types from an arbitrary language.

The first primitive is a generalization of the weakening function from the last section. We modify its specification to allow insertion into any position of a context, not just the beginning. \oplus denotes list concatenation, and it and single-element concatenation $::$ are right associative at the same precedence level.

$$\begin{aligned} \text{weaken} &: \Pi\Gamma_1, \Pi\Gamma_2, \Pi\tau, \text{term } (\Gamma_1 \oplus \Gamma_2) \tau \\ &\rightarrow \Pi\tau', \text{term } (\Gamma_1 \oplus \tau' :: \Gamma_2) \tau \end{aligned}$$

Next is elementwise permutation. We swap the order of two adjacent context elements.

$$\begin{aligned} \text{permute} &: \Pi\Gamma_1, \Pi\Gamma_2, \Pi\tau_1, \Pi\tau_2, \Pi\tau, \text{term } (\Gamma_1 \oplus \tau_1 :: \tau_2 :: \Gamma_2) \tau \\ &\rightarrow \text{term } (\Gamma_1 \oplus \tau_2 :: \tau_1 :: \Gamma_2) \tau \end{aligned}$$

We also want to calculate the free variables of a term. Here we mean those that actually appear, not just those that are in scope. This calculation and related support functions are critical to efficient closure conversion for any language. We write $\mathcal{P}(\Gamma)$ to denote the type of subsets of the bindings in context Γ .

$$\text{freeVars} : \Pi\Gamma, \Pi\tau, \text{term } \Gamma \tau \rightarrow \mathcal{P}(\Gamma)$$

Using the notion of free variables, we can define *strengthening*, which removes unused variable bindings from a context. This operation is also central to closure conversion. An argument of type $\Gamma_1 \subseteq \Gamma_2$ denotes a constructive proof that every binding in Γ_1 is also in Γ_2 .

$$\begin{aligned} \text{strengthen} &: \Pi\Gamma, \Pi\tau, \Pi e : \text{term } \Gamma \tau, \Pi\Gamma', \\ &\text{freeVars } \Gamma \tau e \subseteq \Gamma' \rightarrow \text{term } \Gamma' \tau \end{aligned}$$

We have found these primitive operations to be sufficient for easing the burden of manipulating our higher-level intermediate languages. The derived operations that our system instantiates are: adding multiple bindings to the middle of a context and adding multiple bindings to the end of a context, based on `weaken`; and moving a binding from the front to the middle or from the middle to the front of a context and swapping two adjacent multi-binding sections of a context, derived from `permute`.

6.1 Generic Correctness Proofs

Writing these boilerplate syntactic functions is less than half of the challenge when it comes to proving semantics preservation. The semantic properties of these functions are crucial to enabling correctness proofs for the transformations that use them. We also automate the generation of the correctness proofs, based on an observation about the classes of semantic properties we find ourselves needing to verify for them. The key insight is that we only require that each function *commute with any compositional denotation function* in a function-specific way.

An example should illustrate this idea best. Take the case of the `weakenFront` function for our Source language. Fix an arbitrary denotation function $\llbracket \cdot \rrbracket$ for source terms. We claim that the correctness of a well-written compiler will only depend on the following property of `weakenFront`, written with informal notation: For any Γ, τ , and e with $\Gamma \vdash e : \tau$, we have for any τ' , substitution σ for the variables of Γ , and value v of type $\llbracket \tau \rrbracket$, that:

$$\llbracket \text{weakenFront } \Gamma \tau e \tau' \rrbracket(\sigma, v) = \llbracket e \rrbracket \sigma$$

We shouldn't need to know many specifics about $\llbracket \cdot \rrbracket$ to deduce this theorem. In fact, it turns out that all we need is the standard property used to judge suitability of denotational semantics, compositionality. In particular, we require that there exist functions $f_{const}, f_{var}, f_{app}$, and f_{lam} such that:

$$\begin{aligned} \llbracket n \rrbracket \sigma &= f_{const}(n) \\ \llbracket x \rrbracket \sigma &= f_{var}(\sigma(x)) \\ \llbracket e_1 e_2 \rrbracket \sigma &= f_{app}(\llbracket e_1 \rrbracket \sigma, \llbracket e_2 \rrbracket \sigma) \\ \llbracket \lambda x : \tau. e \rrbracket \sigma &= f_{lam}(\lambda x : \llbracket \tau \rrbracket. \llbracket e \rrbracket(\sigma, x)) \end{aligned}$$

Our generic programming system introspects into user-supplied denotation function definitions and extracts the functions that witness their compositionality. Using these functions, it performs automatic proofs of generic theorems like the one above about `weakenFront`. Every other generated syntactic function has a similar customized theorem statement and automatic strategy for proving it for compositional denotations.

Though space limits prevent us from providing more detail here, we mention that our implementation is interesting in that the code and proof generation themselves are implemented almost entirely inside of Coq's programming language, using dependent types to ensure their soundness. We rely on the technique of reflective proofs [Bou97] to enable Coq programs to manipulate other Coq programs in a type-safe manner, modulo some small “proof hints” provided from outside the logic.

7. Representing Logical Relations

We now turn our attention to formulating the correctness proofs of compiler phases, again using the linearization phase as our example. We are able to give a very simple logical relations argument for this phase, since our meta language CIC is sufficiently expressive to encode naturally the features of both source and target languages. The correctness theorem that we want looks something like the following, for some appropriate type-indexed relation \simeq . For disambiguation purposes, we write $\llbracket \cdot \rrbracket^S$ for Source language denotations and $\llbracket \cdot \rrbracket^L$ for Linear denotations.

```

Fixpoint val_lr (t : sty) : styDenote t
  -> ltyDenote t -> Prop :=
  match t
  return (styDenote t -> ltyDenote t -> Prop) with
  | SNat => fun n1 n2 =>
    n1 = n2
  | SArrow t1 t2 => fun f1 f2 =>
    forall x1 x2, val_lr t1 x1 x2
    -> forall (k : styDenote t2 -> nat),
      exists thrown, f2 x2 k = k thrown
      /\ val_lr t2 (f1 x1) thrown
  end.

```

Figure 3. Coq source code for the first-stage CPS transform’s logical relation

THEOREM 3 (Correctness of linearization). **For** Source term e such that $\Gamma \vdash e : \tau$, if we have valuations σ^S and σ^L for $\llbracket \tau \rrbracket^S$ and $\llbracket \tau \rrbracket^L$, respectively, **such that** for every $x : \tau' \in \Gamma$, we have $\sigma^S(x) \simeq_{\tau'} \sigma^L(x)$, then $\llbracket e \rrbracket^S \sigma^S \simeq_{\tau} \llbracket e \rrbracket^L \sigma^L$.

This relation for values turns out to satisfy our requirements:

$$\begin{aligned} n_1 \simeq_N n_2 &= n_1 = n_2 \\ f_1 \simeq_{\tau_1 \rightarrow \tau_2} f_2 &= \forall x_1 : \llbracket \tau_1 \rrbracket^S, \forall x_2 : \llbracket \tau_1 \rrbracket^L, x_1 \simeq_{\tau_1} x_2 \\ &\rightarrow \exists v : \llbracket \tau_2 \rrbracket^L, \forall k : \llbracket \tau_2 \rrbracket^L \rightarrow \mathbb{N}, \\ &\quad f_2 x_2 k = k v \wedge f_1 x_1 \simeq_{\tau_2} v \end{aligned}$$

We have a standard logical relation defined by recursion on the structure of types. $e_1 \simeq_{\tau} e_2$ means that values e_1 of type $\llbracket \tau \rrbracket^S$ and e_2 of type $\llbracket \tau \rrbracket^L$ are equivalent in a suitable sense. Numbers are equivalent if and only if they are equal. Source function f_1 and linearized function f_2 are equivalent if and only if for every pair of arguments x_1 and x_2 related at the domain type, there exists some value v such that f_2 called with x_2 and a continuation k always throws v to k , and the result of applying f_1 to x_1 is equivalent to v at the range type.

The suitability of particular logical relations to use in compiler phase correctness specifications is hard to judge individually. We know we have made proper choices when we are able to compose all of our correctness results to form the overall compiler correctness theorem. It is probably also worth pointing out here that our denotational semantics are not *fully abstract*, in the sense that our target domains “allow more behavior” than our object languages ought to. For instance, the functions k quantified over by the function case of the \simeq definition are drawn from the full Coq function space, which includes all manner of complicated functions relying on inductive and co-inductive types. The acceptability of this choice for our application is borne out by our success in using this logical relation to prove a final theorem whose statement does not depend on such quantifications.

Once we are reconciled with that variety of caveat, we find that Coq provides quite a congenial platform for defining logical relations for denotational semantics. The \simeq definition can be transcribed quite literally, as witnessed by Figure 3. The set of all logical propositions in Coq is just another type `Prop`, and so we may write recursive functions that return values in it. Contrast our options here with those associated with proof assistants like Twelf [PS99], for which formalization of logical relations has historically been challenging.

8. Proof Automation

Now that we have the statement of our theorem, we need to produce a formal proof of it. In general, our proofs will require significant

manual effort. As anyone who has worked on computerized proofs can tell you, time-saving automation machinery is extremely welcome. In proving the correctness theorems for our compiler, we were surprised to find that a very simple automation technique is very effective on large classes of proof goals that appear. The effectiveness of this technique has everything to do with the combination of typed intermediate languages and our use of dependent types to represent their programs.

As an example, consider this proof obligation that occurs in the correctness proof for linearization.

- **Know:** $\forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_{\Gamma} \sigma^L \rightarrow \exists v : \llbracket \tau_1 \rightarrow \tau_2 \rrbracket^L, \forall k : \llbracket \tau_1 \rightarrow \tau_2 \rrbracket^L \rightarrow \mathbb{N}, \llbracket e_1 \rrbracket^L \sigma^L k = k v \wedge \llbracket e_1 \rrbracket^S \sigma^S \simeq_{\tau_1 \rightarrow \tau_2} v$
- **Know:** $\forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_{\Gamma} \sigma^L \rightarrow \exists v : \llbracket \tau_1 \rrbracket^L, \forall k : \llbracket \tau_1 \rrbracket^L \rightarrow \mathbb{N}, \llbracket e_2 \rrbracket^L \sigma^L k = k v \wedge \llbracket e_2 \rrbracket^S \sigma^S \simeq_{\tau_1} v$
- **Must prove:** $\forall \sigma^S, \forall \sigma^L, \sigma^S \simeq_{\Gamma} \sigma^L \rightarrow \exists v : \llbracket \tau_2 \rrbracket^L, \forall k : \llbracket \tau_2 \rrbracket^L \rightarrow \mathbb{N}, \llbracket e_1 e_2 \rrbracket^L \sigma^L k = k v \wedge \llbracket e_1 e_2 \rrbracket^S \sigma^S \simeq_{\tau_2} v$

It is safe to simplify the goal by moving all of the universal quantifiers and implications that begin it into our proof context as new bound variables and assumptions; this rearrangement cannot alter the provability of the goal. Beyond that, Coq’s standard automation support is stuck. However, it turns out that we can do much better for goals like this one, based on *greedy quantifier instantiation*. Traditional automated theorem provers spend most of their intelligence in determining how to *use* universally-quantified facts and *prove* existentially-quantified facts. When quantifiers range over infinite domains, many such theorem-proving problems are both undecidable and difficult in practice.

However, examining our goal above, we notice that it has a very interesting property: *every* quantifier has a rich type depending on some object language type. Moreover, for any of these types, *exactly one subterm of proof state* (bound variables, assumptions, and goal) that has that type ever appears at any point during the proving process! This observation makes instantiation of quantifiers extremely easy: instantiate any universal assumption or existential goal with *the first properly-typed proof state subterm that appears*.

For instance, in the example above, we had just moved the variables σ^S and σ^L and the assumption $\sigma^S \simeq_{\Gamma} \sigma^L$ into our proof context. That means that we should instantiate the initial σ quantifiers of the two assumptions with these variables and use modus ponens to access the conclusions of their implications, by way of our new assumption. This operation leaves both assumptions at existential quantifiers, so we eliminate these quantifiers by adding fresh variables and new assumptions about those variables. The types of the k quantifiers that we reach now don’t match any subterms in scope, so we stop here.

Now it’s time for a round of rewriting, using rules added by the human user to a hint database. We use all of the boilerplate syntactic function soundness theorems that we generated automatically as left-to-right rewrite rules, applying them in the goal until no further changes are possible. Using also a rewrite theorem expressing the soundness of the \bullet composition operation, this process simplifies the goal to a form with a subterm $\llbracket e_1 \rrbracket^L s^L (\lambda x : \llbracket \tau_1 \rightarrow \tau_2 \rrbracket^L. \dots)$. This lambda term has the right type to use as an instantiation for the universally-quantified k in the first assumption, so, by our greedy heuristic, we make that instantiation. We perform “safe” propositional simplifications on the newly-exposed part of that assumption and continue.

Iterating this heuristic, we discharge the proof obligation completely, with no human intervention. Our very naive algorithm has succeeded in “guessing” all of the complicated continuations needed for quantifier instantiations, simply by searching for

```

(* State the lemma characterizing the effect of
 * the CPS'd term composition operator. *)
Lemma compose_sound : forall (G : list sty)
  (t : sty) (e : lterm G t)
  (t' : sty) (e' : lterm (t :: G) t') s
  (k : _ -> result),
ltermDenote (compose e e') s k
= ltermDenote lin s
  (fun x => ltermDenote lin' (SCons x s) k).
induction e;   (* We prove it by induction on
                  * the structure of
                  * the term e. *)
equation_tac. (* A generic rewriting
                  * procedure handles
                  * the resulting cases. *)
Qed.

(* ...omitted code to add compose_sound to our
 * rewriting hint base... *)

(* State the theorem characterizing soundness of
 * the CPS translation. *)
Theorem cps_sound : forall G t (e : stem G t),
  exp_lr e (cps e).
unfold exp_lr; (* Expand the definition of the
                  * logical relation on
                  * expressions. *)
induction e; (* Proceed by induction on the
                  * structure of the term e. *)
lr_tac.      (* Use the generic greedy
                  * instantiation procedure to
                  * discharge the subgoals. *)
Qed.

```

Figure 4. Snippets of the Coq proof script for the CPS correctness theorem

properly-typed subterms of the proof state. In fact, this same heuristic discharges all of the cases of the linearization soundness proof, once we've stocked the rewrite database with the appropriate syntactic simplifications beforehand. To prove this theorem, all the user needs to do is specify which induction principle to use and run the heuristic on the resulting subgoals.

We have found this approach to be very effective on high-level typed languages in general. The human prover's job is to determine the useful syntactic properties with which to augment those proved generically, prove these new properties, and add them to the rewrite hint database. With very little additional effort, the main theorems can then be discharged automatically. Unfortunately, our lower-level intermediate languages keep less precise type information, so greedy instantiation would make incorrect choices too often to be practical. Our experience here provides a new justification in support of type-preserving compilation.

Figure 4 shows example Coq code for proving the CPS correctness theorem, with a few small simplifications made for space reasons.

9. Further Discussion

Some other aspects of our formalization outside of the running example are worth mentioning. We summarize them in this section.

9.1 Logical Relations for Closure Conversion

Formalizations of closure conversion and its correctness, especially those using operational semantics, often involve existential types and other relatively complicated notions. In our formalization, we are able to use a surprisingly simple logical relation to characterize closure conversion. It relates denotations from the CPS and CC languages, indicated with superscripts P and C , respectively. We lift various definitions to vectors in the usual way.

$$\begin{aligned} n_1 \simeq_N n_2 &= n_1 = n_2 \\ f_1 \simeq_{\vec{\tau} \rightarrow N} f_2 &= \forall \vec{x}_1 : [\vec{\tau}]^P, \forall x_2 : [\vec{\tau}]^C, x_1 \simeq_{\vec{\tau}} x_2 \\ &\quad \rightarrow f_1 x_1 \simeq_{\tau_2} f_2 x_2 \end{aligned}$$

This relation is almost identical to the most basic logical relation for simply-typed lambda calculus! The secret is that our meta language CIC “has native support for closures.” That is, Coq's function spaces already incorporate the appropriate reduction rules to capture free variables, so we don't need to mention this process explicitly in our relation.

In more detail, the relevant denotations of CC types are:

$$\begin{aligned} [\vec{\tau} \rightarrow N] &= [\tau_1] \times \dots \times [\tau_n] \rightarrow N \\ [\vec{\tau}^1 \times \vec{\tau}^2 \rightarrow N] &= ([\tau_1^1] \times \dots \times [\tau_n^1]) \rightarrow ([\tau_1^2] \times \dots \times [\tau_m^2]) \rightarrow N \end{aligned}$$

Recall that the second variety of type shown is the type of code pointers, with the additional first list of parameters denoting the expected environment type. A CPS language lambda expression is compiled into a packaging of a closure using a pointer to a fresh code block. That code block will have some type $\vec{\tau}_1 \times \vec{\tau}_2 \rightarrow N$. The denotation of this type is some meta language type $T_1 \rightarrow T_2 \rightarrow N$. We perform an immediate partial application to an environment formed from the relevant free variables. This environment's type will have denotation T_1 . Thus, the partial application's type has denotation $T_2 \rightarrow N$, making it compatible with our logical relation. The effect of the closure packaging operation has been “hidden” using one of the meta language's “closures” formed by the partial application.

9.2 Explicating Higher-Order Control Flow

The translation from CC to Alloc moves from a higher-order, terminating language to a first-order language that admits non-termination. As a consequence, the translation correctness proof must correspond to some explanation of why CC's particular brand of higher-order control flow leads to termination, and why the resulting first-order program returns an identical answer to the higher-order original.

The basic proof technique has two main pieces. First, the logical relation requires that any value with a code type terminates with the expected value when started in a heap and variable environment where the same condition holds for values that should have code type. Second, we take advantage of the fact that function definitions occur in dependency order in CC programs. Our variable binding restrictions enforce a lack of cycles via dependent types. We can prove by induction on the position of a code body in a program that any execution of it in a suitable starting state terminates with the correct value. Any code pointer involved in the execution either comes earlier in the program, in which case we have its correctness by the inductive hypothesis; or the code pointer has been retrieved from a variable or heap slot, in which case its safety follows by an induction starting from our initial hypothesis and tracking the variable bindings and heap writes made by the program.

9.3 Garbage Collection Safety

The soundness of the translation from Flat to Asm depends on a delicate safety property of well-typed Flat programs. It is phrased in terms of the register typings we have available at every program

point, and it depends on the definition of pointer isomorphism we gave in Section 2.7.

THEOREM 4 (Heap rearrangement safety). *For any register typing Δ , abstract heaps m_1 and m_2 , and register files R_1 and R_2 , if:*

1. *For every register r with $\Delta(r) = \text{ref}$, $R_1(r)$ is isomorphic to $R_2(r)$ with respect to m_1 and m_2 .*
2. *For every register r with $\Delta(r) \neq \text{ref}$, $R_1(r) = R_2(r)$.*

and p is a Flat program such that $\Delta \vdash p$, we have $\llbracket p \rrbracket R_1 m_1 = \llbracket p \rrbracket R_2 m_2$.

This theorem says that it is safe to rearrange a heap if the relevant roots pointing into it are also rearranged equivalently. Well-typed Flat programs can't distinguish between the old and new situations. The denotations that we conclude are equal in the theorem are traces, so we have that valid Flat programs return identical results (if any) and make the same numbers of function calls in any isomorphic initial states.

The requirements on the runtime system's new operation allow it to modify the heap arbitrarily on each call, so long as the new heap and registers are isomorphic to the old heap and registers. The root set provided as an operand to new is used to determine the appropriate notion of isomorphism, so the heap rearrangement safety theorem is critical to making the correctness proof go through.

9.4 Putting It All Together

With all of the compiler phases proved, we can compose the proofs to form a correctness proof for the overall compiler. We give the formal version of the theorem described informally in the Introduction. We use the notation $T \downarrow n$ to denote that trace T terminates with result n .

THEOREM 5 (Compiler correctness). *Let $m \in \mathbb{H}$ be a heap initialized with a closure for the top-level continuation, let p be a pointer to that closure, and let R be a register file mapping the first register to p . Given a Source term e such that $\cdot \vdash e : \mathbb{N}, \llbracket e \rrbracket R m \downarrow \llbracket e \rrbracket()$.*

Even considering only the early phases of the compiler, it is difficult to give a correctness theorem in terms of equality for all source types. For instance, we can't compose parametrically the results for CPS transformation and closure conversion to yield a correctness theorem expressed with an equality between denotations. The problem lies in the lack of full abstraction for our semantics. Instead, we must use an alternate notion of equality that only requires functions to agree at arguments that are denotations of terms. This general notion also appears in the idea of pre-logical relations [HS99], a compositional alternative to logical relations.

10. Implementation

The compiler implementation and documentation are available online at:

<http://ltamer.sourceforge.net/>

We present lines-of-code counts for the different implementation files, including proofs, in Figure 5.

These figures do not include 3520 lines of Coq code from a programming language formalization support library that we have been developing in tandem with the compiler. Additionally, we have 2716 lines of OCaml code supporting generic programming of syntactic support functions and their correctness proofs. Developing these re-usable pieces was the most time-consuming part of our overall effort. We believe we improved our productivity an order of magnitude by determining the right language representation

File	LoC
Source	31
...to...	116
Linear	56
...to...	115
CPS	87
...to...	646
CC	185
...to...	1321
Alloc	217
...to...	658
Flat	141
...to...	868
Asm	111

File	LoC
Applicative dictionaries	119
Traces	96
GC safety	741
Overall compiler	119

Figure 5. Project lines-of-code counts

scheme and its library support code, and again by developing the generic programming system. With those pieces in place, implementing the compiler only took about one person-month of work, which we feel validates the general efficacy of our techniques.

It is worth characterizing how much of our implementation must be trusted to trust its outputs. Of course, the Coq system and the toolchain used to compile its programs (including operating system and hardware) must be trusted. Focusing on code specific to this project, we see that, if we want only to certify the behavior of particular output assembly programs, we must trust about 200 lines of code. This figure comes from taking a backwards slice from the statement of any individual program correctness theorem. If we want to believe the correctness of the compiler itself, we must add additionally about 100 lines to include the formalization of the Source language as well.

11. Related Work

Moore produced a verified implementation of a compiler for the Piton language [Moo89] using the Boyer-Moore theorem prover. Piton did not have the higher-order features that make our present work interesting, and proofs in the Boyer-Moore tradition have fundamentally different trustworthiness characteristics than Coq proofs, being dependent on a large reasoning engine instead of a small proof-checking kernel. However, the Piton work dealt with larger and more realistic source and target languages.

The VLISP project [GRW95] produced a Scheme system with a rigorous but non-mechanized proof of correctness. They also made heavy use of denotational semantics, but they dealt with a dynamically-typed source language and so did not encounter many of the interesting issues reported here related to type-preserving compilation.

Semantics preservation proofs have been published before for individual phases of type-preserving compilers, including closure conversion [MMH95]. All of these proofs that we are aware of use operational semantics, forfeiting the advantages of denotational semantics for mechanization that we have shown here.

Recent work has implemented tagless interpreters [PTS02] using generalized algebraic datatypes and other features related to dependent typing. Tagless interpreters have much in common with our approach to denotational semantics in Coq, but we are not aware of any proofs beyond type safety carried out in such settings.

The CompCert project [Ler06] has used Coq to produce a certified compiler for a subset of C. Because of this source language choice, CompCert has not required reasoning about nested variable scopes, first-class functions based on closures, or dynamic allocation. On the other hand, they deal with larger and more realistic source and target languages. CompCert uses non-dependently-

typed abstract syntax and operational semantics, in contrast to our use of dependent types and denotational semantics; and we focus more on proof automation.

Many additional pointers to work on compiler verification can be found in the bibliography by Dave [Dav03].

12. Conclusion

We have outlined a non-trivial case study in certification of compilers for higher-order programming languages. Our results lend credence to the suitability of our implementation strategy: the encoding of language syntax and static semantics using dependent types, along with the use of denotational semantics targeting a rich but formalized meta language. We have described how generic programming and proving can be used to ease the development of type-preserving compilers and their proofs, and we have demonstrated how the certification of type-preserving compilers is congenial to automated proof.

We hope to expand these techniques to larger and more realistic source and target languages. Our denotational approach naturally extends to features that can be encoded directly in CIC, including (impredicative) universal types, (impredicative) existential types, lists, and trees. Handling of “effectful” features like non-termination and mutability without first compiling to first-order form (as we’ve done here in the later stages of the compiler) is an interesting open problem. We also plan to investigate further means of automating compiler correctness proofs and factorizing useful aspects of them into re-usable libraries.

There remains plenty of room to improve the developer experience in using our approach. Programming with dependent types has long been known to be tricky. We implemented our prototype by slogging through the messy details for the small number of features that we’ve included. In scaling up to realistic languages, the costs of doing so may prove prohibitive. Some recently-proposed techniques for simplifying dependently-typed Coq programming [Soz06] may turn out to be useful.

Coercions between different dependent types appear frequently in the approach we follow. It turns out that effective reasoning about coercions in type theory requires something more than the computational equivalences that are used in systems like CIC. In Coq, this reasoning is often facilitated by adding an axiom describing the computational behavior of coercions. Ad-hoc axioms are a convenient way of extending a proof assistant’s logic, but their loose integration has drawbacks. Coq’s built-in type equivalence judgment, which is applied automatically during many stages of proof checking, will not take the axioms into account. Instead, they must be applied in explicit proofs. New languages like Epigram [MM04] design their type equivalence judgments to facilitate reasoning about coercions. Future certified compiler projects might benefit from being developed in such environments, modulo the current immaturity of their development tools compared to what Coq offers. Alternatively, it is probably worth experimenting with the transplantation of some of these new ideas into Coq.

Acknowledgments

Thanks to Manu Sridharan and the anonymous referees for helpful comments on drafts of this paper.

References

- [ABF⁺05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In *Proc. TPHOLs*, pages 50–65, 2005.
- [BC04] Yves Bertot and Pierre Castérán. *Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
- [Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Proc. STACS*, pages 515–529, 1997.
- [Dav03] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.
- [dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
- [Gim95] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Proc. TYPES*, pages 39–59. Springer-Verlag, 1995.
- [GRW95] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.
- [HS99] Furio Honsell and Donald Sannella. Pre-logical relations. In *Proc. CSL*, pages 546–561, 1999.
- [Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54, 2006.
- [MM04] Conor McBride and James McKinna. The view from the left. *J. Functional Programming*, 14(1):69–111, 2004.
- [MMH95] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. Technical Report CMU-CS-FOX-95-05, Carnegie Mellon University, 1995.
- [Moo89] J. Strother Moore. A mechanically verified language implementation. *J. Automated Reasoning*, 5(4):461–492, 1989.
- [MSLL07] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI*, 2007.
- [MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
- [PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. PLDI*, pages 199–208, 1988.
- [Plo73] G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, University of Edinburgh, 1973.
- [Plo75] Gordon D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. CADE*, pages 202–206, 1999.
- [PTS02] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *Proc. ICFP*, pages 218–229, 2002.
- [She04] Tim Sheard. Languages of the future. In *Proc. OOPSLA*, pages 116–119, 2004.
- [Soz06] Matthieu Sozeau. Subset coercions in Coq. In *Proc. TYPES*, 2006.
- [TMC⁺96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *Proc. PLDI*, pages 181–192, 1996.

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/221554772>

Promises: Limited Specifications for Analysis and Manipulation.

Conference Paper in Proceedings - International Conference on Software Engineering · January 1998

DOI: 10.1109/ICSE.1998.671113 · Source: DBLP

CITATIONS

49

READS

25

3 authors:



Edwin C. Chan
Carnegie Mellon University

3 PUBLICATIONS 608 CITATIONS

[SEE PROFILE](#)



John Tang Boyland
University of Wisconsin - Milwaukee

74 PUBLICATIONS 1,449 CITATIONS

[SEE PROFILE](#)



William L. Scherlis
Carnegie Mellon University

80 PUBLICATIONS 4,940 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Upgrading SASyLF [View project](#)

Promises: Limited Specifications for Analysis and Manipulation*

Edwin C. Chan

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA 15213
+1 412 268 3076
chance@cs.cmu.edu

John T. Boyland

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA 15213
+1 412 268 3738
John.Boyland@acm.org

William L. Scherlis

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA 15213
+1 412 268 8741
scherlis@cs.cmu.edu

ABSTRACT

Structural change in a large system is hindered when information is missing about portions of the system, as is often the case in a distributed development process. An annotation mechanism called *promises* is described for expressing properties that can enable many kinds of structural change in systems. Promises act as surrogates for an actual component, and thus are analogous to "header" files, but with more specific semantic information. Unlike formal specifications, however, promises are designed to be easily extracted from systems and managed by programmers using automatic analysis tools. Promises are described for effects, unique references, and use properties. By using promises, a component developer can offer additional opportunity for change (flexibility) to clients, but at a potential cost in flexibility for the component itself. This suggests the possibility of using promises as a means to allocate flexibility among the components of a system.

KEYWORDS

Aliasing, effects, program restructuring, headers, implementation flexibility, limited references, effect regions, separate compilation, unique references

*This material is based upon work supported by the National Science Foundation under Grant No. CCR-9504339 and by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, the National Science Foundation, or the U.S. Government.

©1998 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

20th International Conference on Software Engineering (ICSE '98), April 19–25, Kyoto, Japan.

0-8186-8368-6/98 \$10.00 © 1998 IEEE

1 INTRODUCTION

Analysis and alteration of a part of a software system is hindered when information is missing about the remaining portions of the system and conservative assumptions must be made. We describe a language of limited specifications, called *promises*, to express properties that are significant in making structural change in systems, but that also can be easily extracted from program components with automatic tools playing a primary role.

Promises offer a compromise between the minimal type information managed in "header" files and the more extensive semantic information that might be in an architectural specification. The additional semantic information provided by the promise annotations can yield more precise analyses, and hence enable a broader range of program manipulation and restructuring to be carried out with a guarantee of soundness. Structural modifications are common in software evolution; examples include restructuring a class hierarchy, copying an encapsulation for tailoring to a specialized subset of uses, changing data representation, and abstracting a new function definition from existing code.

Promises are intended to be sufficiently easy to specify and validate that they can be used in routine design and programming activity, particularly program evolution using tool assistance. In this regard, promises are similar to the C-language annotations of tools such as LCLint [7]. The LCLint annotations focus on a different set of goals—specification of encapsulations and error-detection. For example, LCLint provides an annotation language for the explicit specification of programmer-intended abstract data type boundaries, which are not directly expressible in the C language. Annotations such as these support analysis algorithms that perform error detection such as ensuring that the encapsulations are respected.

In this paper we define portions of a language of promises for Java. The promises relate to effects, alias-

ing, and use properties, and can be attached to a variety of different program elements. The promises enable a component implementor to offer “contracts” to clients that permit clients to make structural changes that might not be sound without the guarantee provided by the promise. For example, guarantees concerning the effects of a method may enable safe reordering of computation. A component implementor may require promises of the clients as well—we call these requirements *demands*. Our thesis is that there is a relatively small set of promise types needed to enable a broad range of restructuring manipulations to be carried out safely, that the introduction of promises into code can be largely automated, and that promises can be effectively managed and changed by programmers and tools. In general, however, promises cannot feasibly be checked at runtime. This regime of promises is being incorporated into an experimental tool that supports systematic source-level structural change in Java programs. This will enable us to explore the practicality of this approach for realistic case studies.

A key design issue is how to formulate promises that are useful to component clients but do not unduly restrict choices in component implementation. For example, promises concerning effects indicate what portions of mutable state can be read or modified. Specifying effects directly in terms of fields (specific variables in classes) may unnecessarily expose implementation details and restrict subclasses. Instead, we employ the concept of abstract regions of an object, onto which the fields are mapped. This enables useful local analysis while retaining implementation flexibility.

In the Section 2 below, we introduce promises relating to effects, aliases, and structure. While promises about a component grant additional flexibility to its clients, they also limit flexibility for evolution of the component itself. This suggests that, with appropriate tools, a (mature) regime of promises could serve as a “currency of flexibility” in the management of larger software systems, with use (or non-use) of promises providing a mechanism to “allocate” engineering flexibility among the elements of a system in response to extrinsic engineering factors. In Section 3 we consider some technical issues concerning the feasibility of this prospect, particularly the role of tools in managing promises.

2 Promises

Promises are supra-linguistic formal annotations to programs. That is, they have precise meaning, but are not part of the actual programming language definition. They are meant to be interpreted by program analysis and manipulation tools and (generally) ignored by compilers. Annotations can be applied to several different kinds of abstract-syntactic types. They can be made

syntactically concrete as formally structured comments (as in Anna, Javadoc, and LCLint) or by direct integration into the syntax (and then presumably removed by a preprocessor to the compiler). For convenience in presentation, we use the latter approach in this paper. (Regardless, tools can be used to display promises to a programmer using any convenient syntactic form, even eliding promises not of current interest.)

Semantically, promises about a component restrict choices for its implementation and evolution (once clients rely on these promises). But in client code, these restrictions have the opposite effect—they enable more precise analysis, and hence they enable a broader range of meaning-preserving changes in client code.

In many software engineering processes, the kinds of information expressed in a promise may normally be part of design documents or otherwise be expressed in organizational memory. There are obvious advantages to capturing a promise explicitly: The representation is formal and precise, and it is in simple programmer-oriented language. Tools can interpret the promise and provide analysis support, for example to validate hypotheses programmers may have about a system, or to suggest new promises that could be offered. And tools can exploit promise information to help a software engineer carry out larger-scale structural change not otherwise feasible particularly on incomplete programs [10, 17, 19].

As with other kinds of specifications, however, there are costs—each promise must be identified and expressed. Nonetheless, we suggest that promises differ from full functional specifications in two important respects. First, the limited semantic content of promises makes them easier to state, validate, and exploit. Second, like LCLint, promises offer an incremental approach to adoption by software engineers—identifying and expressing an individual promise is a very small increment of effort, that, in general, should yield some increment of value in client flexibility, analysis results, and manipulations enabled. Promises enable software engineers to be selective about both the specific elements of a system that they annotate and the extent of promises made for those elements.

Our selection of particular semantic properties to express as promises is guided by the analyses needed to enable a wide range of evolutionary changes in programs, particularly changes to shared abstractions in larger software systems. “Code rot” can be considered to be the persistence of abstractions beyond their time. Change to abstractions can be facilitated when they are better encapsulated, when more of their properties are directly expressed, and when their uses are more precisely understood. For this reason, promises are *not* de-

signed to address functional properties, but rather the more forgettable “bureaucratic” ones—semantic properties such as effects and mutability, and structural properties such as where the call sites of a method are located.

Feasibility of the overall approach requires that promises be easily expressed and understood by programmers, and that, with use of appropriate tools, promises be easy to offer (extract directly from code) and validate (when explicitly claimed). The three classes of promises considered in this paper can, in general, be extracted from code using automatic tools with a limited amount of programmer guidance. Validation of promises claimed for a component (or explicitly sought by a client) is also straightforward. Validation is done by analysis; runtime checks are not practical for most of the promises we consider (see Figure 1).

An example of a simple restructuring manipulation frequently carried out by a programmer is the abstraction of some existing fragment(s) of code into a new function definition (e.g. a method declaration in Java). This might be done simply to reuse those code fragments elsewhere in the program, or done as part of a more complex manipulation, such as redistributing code while changing the internal representation of a class. Carrying out the abstraction requires identifying the code site of the existing code to abstract as the body of the new method definition, the sites of contained code fragments that will remain at the calling site as actual parameters, a specification of the order in which the parameters will appear, a site for the new definition, and names for the new method and its formal parameters. Assurance of soundness of this seemingly simple manipulation requires several analyses that are supported by semantic and structural promises. For instance, both order and frequency of computation may be affected by abstracting some computation into an actual parameter. Effects promises, as described in Section 2.1 and made more precise in Section 2.2, allow the determination of safe reorderings. Name bindings may be affected by the relocation of code to a new site (e.g., across an encapsulation or scope boundary) and by the introduction of new formal parameter names and a new name for the abstracted method. Section 2.3 describes a way to identify the uses of the class whose interface is being changed by the introduction of this new method. Together, these three kinds of promises can be used to guarantee soundness of an application of this manipulation, as well as others.

2.1 Effects analysis

The abstraction of a code fragment into a new method definition is one of many cases where reordering of code is required in order to enable a more significant struc-

<code>reads, modifies</code>	Possible effects of methods.
<code>unique, unshared</code>	Unaliased references.
<code>limited</code>	Reference not to be stored.
<code>immutable(*)</code>	Referenced object is never modified.
<code>instanceof(*)</code>	Referenced object is an instance of one of specified concrete classes.
<code>usedBy</code>	Clients of a field, method, or class.

Figure 1: Some promises and demands (* = not discussed in this paper).

tural change. The analysis primarily involves identifying effects and dependencies on them. Effects, for our purposes, include reading and writing of modifiable store, since modeling of data dependencies requires us to track “read effects” as well as write effects.

Consider the following class that encapsulates a variable:

```
class Var
{ private int val = 0;
  public void set(int x) { val = x; }
  public int get() { return val; } }
```

Suppose, in our method abstraction scenario, we have a block that contains an expression `v.get()`, which may in a conditional or a loop, and we wish to abstract the block to create a new method `M`, but with the expression `v.get()` remaining behind as an actual parameter in the call that replaces the block. Let `v` be non-null (i.e., it points to an allocated object). Then it appears that as long as nothing else in the block modifies the private instance variable in `v` (for example if we know `v` is not aliased anywhere), it would be safe to extract the expression. This information is not sufficient, however: `v` may in fact be an instance of a *subclass* of `Var`, perhaps of a class such as the following:

```
class NoisyVar extends Var
{ public int get()
  { System.out.println("Var is read!");
    return super.get(); } }
```

This `get` method has an effect. If `v.get()` were in a conditional or inside a loop, then computing it in advance as a parameter could lead to extra output or missing output. We could designate `Var` as a final class (and this designation is a kind of promise). But if subclasses to `Var` are to be allowed, we need to be able to offer a commitment to clients of `Var` that its subclasses don’t have any more effects than `Var` has.

In this example, we concretely express the promises by adding extra modifiers to method headers (see prior

discussion concerning syntax). We annotate `Var` with `reads` and `modifies` promises in the style of Java `throws` declarations:

```
class Var
{ private int val = 0;
  public void set(int x) modifies val
  { val = x; }
  public int get() reads val
  { return val; } }
```

The `reads` promise denotes that the method can access `val` but have no other effects. The `modifies` promise indicates that `set` may read and write `val`, but no other variables. As with `throws`, we require these promises to be respected by overriding methods as well. If the class `NoisyVar` were added to the system, the tool would detect the lack of conformance in the overriding `get` method, and alert the developer. The `reads` and `modifies` promises thus allow clients of `Var` to reason about the effects of method calls without needing to know the bodies of the methods, as would be the case in a distributed development process.

Unfortunately, there are two problems with this approach:

- The names of private instance variables are revealed in the headers of methods, violating the encapsulation.
- Subclasses may be unnecessarily restricted.

For an example of the second problem consider the following class:

```
class UndoableVar extends Var
{ private int saved = 0;
  public void set(int x) modifies val, saved
  { saved = get();
    super.set(x); }
  public void undo() modifies val, saved
  { set(saved); } }
```

Here `set` does not obey the restriction of the method that it overrides and thus would not be legal. Nonetheless, the additional effects do not affect whether an expression such as `v.get()` can be reordered with respect to an expression such as `w.set(42)`.

Thus instead of listing instance variables, effects promises specify *regions* that are read or modified. Regions are abstractions for sets of fields. A class may have multiple regions each of which refers to a disjoint set of instance variables. Figure 2 shows how `Var` and `UndoableVar` are annotated with regions. Since they are abstract, regions permit information hiding and offer control over implementation flexibility. Because they are disjoint, regions permit effects analysis to determine safe reorderings.

```
class Var
{ region Value;
  private int val in Value = 0;
  public void set(int x) modifies Value
  { val = x; }
  public int get() reads Value
  { return val; } }

class UndoableVar extends Var
{ private int saved in Value = 0;
  public void set(int x) modifies Value
  { saved = get();
    super.set(x); }
  public void undo() modifies Value
  { set(saved); } }
```

Figure 2: An example of effects promises

In Java, fields may be declared “static,” that is, shared by all objects of a class. Similarly, regions may be declared “static,” in which case the determination of effect overlap takes the sharing into account.

The `reads` and `modifies` promises describe the effects of method calls. A method may have effects on objects other than the receiver; it may read or write objects passed as parameters, objects available in public static fields (for example `System.out`), or objects indirectly reached through one of these other objects. In the `reads` and `modifies` promises, $p.r$ refers to region r of the object passed as parameter p , $C.s.r$ refers to region r of the object in public static field $C.s$, and $C.r$ refers to region r of *any* object (or set of objects) of class C . The latter notation involves a loss of precision. The shorthand “ $*$ ” refers to all regions of the object. Analogously, $p.*$, $C.s.*$ and $C.*$ refer to all regions of (respectively) an object passed as a parameter, an object in a public static field, and any object of class C . Finally, the region $**$ (with the same meaning as `Object.*`) refers to all regions of all objects. If a method does not declare any effects (and analysis has not been done to infer effects promises) then the most conservative approximation is assumed, which is that it modifies $**$.

Effects annotations can be verified using intra-procedural analysis (that is, analysis local to a method body). The process is very similar to that used by compilers to check `throws` clauses in Java. One first determines the local effects (reads and writes to fields) and then adds the effects of every method call in the body using the annotations on the statically selected methods. The resulting set of effects is then compared with the effects declared for the method. Effects can be ignored if they occur on newly created objects. More generally, the existence of “unique” (unaliased) references

(as explained shortly) can make effects inference more precise.

Furthermore, once a developer has specified the regions for each field, an *inter-procedural* analysis can be used to infer the effects of a large body of code all at once and get a system of promises in place. The technique outlined above computes the effects of each method in terms of the effects of other methods. A set of constraints can thus be induced from the method bodies, and the least fixed point of these constraints will give a valid set of promises. Some of the resulting effects promises may prove too strict for subsequent development, and may need to be liberalized.

2.2 Unique References

As shown above, effect specifications are useful in assuring soundness of manipulations that require reordering code. If two effects take place in distinct regions (or in regions of distinct objects), they cannot conflict. Observe that if the effects take place in a region of an object that has just been allocated, they are invisible to the caller and need not be declared in an effects promise. This is because the “just allocated” objects are *unique references*—unaliased references that cannot be accessed in any other context. In this section, we formalize and generalize this concept into a separate kind of promise.

A promise that a reference is unique also ensures that the object to which it points can be copied without disturbing the meaning of the program (because object identity is irrelevant). This scenario may occur as part of a manipulation that copies a class to make two specialized classes. At some point, one may need to treat an object of one derived class as one of the other class. If the reference to the object is unique, the contents of the object can be safely copied into a new object of the other class and the original object abandoned.

Suppose we wanted to swap the order of the two calls to `Pair.copy()` in the following example:

```
class Pair
{ Object a, b;
  static void copy(Pair dst, Pair src)
    reads src.* modifies dst.* { ... } }
class Main
{ static void main(Pair a, Pair b, Pair c)
  { copy(b, a);
    copy(new Pair(), c); } }
```

Looking at its region specification, we may know that the first call, `copy(b, a)`, could modify `b`. In the absence of other information, a conservative analysis forces us to assume the worst, which is that it could also change `a`, or any other object of class `Pair` (or its subclasses), because any of those could be aliases of `b`. Sim-

ilarly, the same call could also read `a` and any of its aliases. In other words, we could not safely reorder the two statements.

If all references are known to have no aliases, however, there is no confounding, we can achieve maximal precision, and the reordering change is sound. One obvious way to achieve that situation is to never allow any aliases, but, in general, that restriction is impractical (but see Baker’s Linear Lisp [3]). Still, these insights motivate the concepts behind *unique* parameters and return values. References (other than `null`) passed as such parameters or returned from methods or constructors so declared can thus be counted on to be distinct from any reference stored in another variable or returned by any method. This is an aggressive approach to alias control that facilitates many structural manipulations, particularly those that involve introduction and elimination of copying, and the introduction of destructive operations on data structures.

A reference is *unique* when it is initially created, and also when it is the last live reference to a unique object passed as a parameter or returned from a method or constructor. (Note that while a unique reference is unaliased, references in fields of the referenced object might not themselves be unique.) Observe that a *unique* annotation on a *parameter* is in fact a *demand*—a promise required of each client.

The return value of a method is declared unique in the following manner:

```
unique Var makeUniqueVar()
{ Var v = new Var();
  ...
  return v; }
```

This method creates a unique reference, performs some other tasks and then returns it. As long as the other tasks do not create lasting aliases, uniqueness is preserved. There is a potential problem with `makeUniqueVar`: the value returned by `new Var()` need not be unique, since the reference could be compromised by any of the various constructors that were called in the process. Looking at the definition of the class `Var`, we note that its constructor does not create an alias to the reference (`this`) it gets from the constructor for `java.lang.Object`. Thus, we should annotate it with a promise that the reference returned will be unique.

We do not annotate *local variables* with ‘unique’ promises, because such annotations would seem to preclude harmless local aliases, and, more seriously, because a local variable may only be unique during part of its lifetime. Thus, at each execution point, the tool keeps a record of which locals contain references that have no external aliases and for each of these ‘externally

'unique' locals, which other locals may alias it.

Passing a unique reference as an actual parameter (including implicitly using an instance method of the unique reference) may involve the loss of uniqueness since the called method may introduce lasting aliases. (In Java, objects are not copied when they are passed as parameters; method calls induce the creation of aliases.)

We can, however, take advantage of the fact that the scope changes on transfer of control from caller to callee, and weaken our notion of 'aliasing' by making it relative to what is in scope and accessible to a method. This insight suggests we can allow unique references as parameters as long as the method promises that it will not create lasting aliases of the reference. This restriction is accomplished by designating the formal parameter as a *limited* reference parameter. (Observe that when a method's formal parameter is `limited`, this is a promise, not a demand.) Limited references may not be assigned to a field either directly or indirectly through another method, and cannot be returned from methods. The first constraint is to keep from creating globally accessible (and potentially long-lasting) aliases, while the second is to keep from introducing 'unknown' aliases in calling frames. Declaring a *method* to be limited has the effect of declaring the receiver `this` to be limited. If the receivers for `get` and `set` are limited, then we can declare a method such as the following:

```
public static void bump(limited Var v)
    modifies v.Value
{ v.set(v.get()+1); }
```

Since the parameter is limited, this method can be called with a unique reference without creating an alias, and the unique reference will remain unique on return from the method. Thus `bump` could be called in the body of `makeVar`.

When we pass a unique reference to a method as limited parameter, the effects analysis of the called method cannot use the fact that the reference is unaliased. But since the caller's copy of the reference cannot be accessed during the dynamic context of the call, one can pass a unique reference as a *limited unique* reference instead (as long as the reference isn't passed as more than that one parameter). Limited unique parameters are unaliased within a certain dynamic context and have the properties of both unique and limited parameters. As with unique references, limited unique references may be passed to methods as either limited or limited unique parameters.

Up until this point, we have only considered parameters and return values as unique. Although useful in their own right, they can only store unique references temporarily until they go out of scope 'forever', and so we

need more permanent places for them, which are *unshared fields*. Unshared fields are really no more than a form of unique variable, except allocated in the store as part of an object. Similarly *static unshared* fields hold unique references associated with the class object.

A problem with unshared fields is that they invalidate the assumption that a variable, and thus the reference it contains, is only accessible from a single frame. For instance, suppose that within a class with an unshared field, we call a method with the reference from this field as a parameter. Since there is no guarantee that the object with the unshared field is itself unique, we cannot assume the field will be inaccessible during the call.

A simple solution to this is to use a promise that guarantees that the code being called will not access that field. Thus, we only allow an unshared field to be an actual parameter if it is passed as limited and the method to which it is passed does not read or modify the region for that field (that is, the method must have the appropriate reads and modifies promises). As a consequence, if we have existing code that does need to access that region, we may have to nullify the unshared field during the call.

A tool can infer `limited` promises for method parameters, and `unique` promises for the return value: the tool starts with tentative `limited` promises for each parameter and local variable, a `unique` promise for the return value. Then the tool checks the promises, removes any that cannot be verified, repeating until the method body checks. When the verification finally concludes successfully (at worst when all the tentative promises have been removed), any remaining promises are verifiable, by definition.

The uniqueness of parameters must be inferred in a different way, because rather than limiting the implementation of the method, uniqueness of the parameters limits the callers (i.e., they are demands). If all the call sites of the method are known (using the `usedBy` promise explained in the following section), the actual parameters can be examined for whether they are unique references (or perhaps limited unique references) and if all are, the formal parameter can be given the appropriate annotation.

Inference of `limited` and `unique` promises can be performed on a group of methods after the developer has identified unshared fields. First the tool tentatively assumes the most restrictive scenario: all parameters are limited, all return values unique, all local variables limited, and finally a `unique` promise for parameters of methods for which all call sites are known. All methods are checked and any unverifiable tentative promises are discarded, iterating until all promises can be verified. If effects promises are also to be inferred, they should be

inferred once the uniqueness promises are in place, because uniqueness makes effects inference more precise.

This inference method can reduce the work of adding promises, but may fail to come up with useful annotations when the code deviates slightly from the rules required of limited or unique values. In this case, the developer may first make a few promises in key places, then perform the inference, examine where the hand-added promises fail to verify, and iterate.

2.3 Structure

Changes to the interface or functionality of a component is likely to require concomitant changes to client code. In a small scale development environment where a single person or a small team controls all the source code, all the affected code can be identified. When the system is made up of separate subsystems that are built by different teams or obtained as binaries from a third-party the situation becomes problematic. It may be difficult for a developer to be aware even of the extent of usage of a component, much less its nature. This lack of information may make changes to a class impossible without risking the soundness of the system.

Making a complete-program (i.e., closed-world) assumption implies that all use-sites are contained in the code provided. Many analysis techniques require this assumption, and this limits their value. Promises offer a mechanism of surrogacy that enables analyses on incomplete systems. (In this regard there is some analogy with the internal annotation that occurs in some complete-program inter-procedural analyses.)

Our approach to this brittleness problem has two elements. First, the sort of semantic promises explored in the previous two sections can enable soundness to be assured for many structural and semantic changes that would otherwise be too risky in the absence of the complete program text. Second, we introduce an additional form of promise that helps define the scope of usage of a component more precisely than through the usual visibility attributes (`private`, `protected`, and so on) in the programming language. By explicitly annotating declarations with information about their use-sites, we can in many cases regain this sense of closure by bounding the amount of code that may require analysis and modification.

For Java, this means that we annotate classes, fields, and methods with enumerations of all the classes (other than the owning class) that could contain a use-site. Such promises need only be added if and when they are needed in order to enable manipulations. Any declaration can be annotated with a “used by” list of the following form: `UsesList := usedBy (ClassName)*`

Directly implemented, this scheme would likely be

overly cumbersome, since both the number and size of annotations will increase significantly with the size of a system. In addition, small changes to code may necessitate large-scale changes in promises throughout the system.

Annotations would need to be updated each time a new client class began to use a component class. But annotations would not need to be changed when further usesites are added within an existing client, nor would they need to change if any of those classes stopped accessing those fields (though the resulting promises would then be more conservative than necessary and a tool might request source access to the superfluous classes).

A two-fold refinement addresses these difficulties: we use abstract sets of client classes, and also use Java’s package and inheritance structure to implicitly limit uses. First, rather than repeatedly specifying a set of classes, the set can be given a name:

```
SetDecl  :=  usedSet SetName = UsedSet ;
UsedSet  :=  (ClassName|SetName) *
UsesList :=  usedBy UsedSet
```

A set name thus denotes the union of the explicitly-mentioned classes and sets.

By employing used sets, the task of writing promises can be factored, and this task can be supported by tools, with developer involvement in managing the tradeoff between the cost of the annotation process and the precision obtained.

Observe that Java already has named sets, called packages, which are implicitly created by identifying a class as an element, but which are never explicitly enumerated.

For example, if the class `fluid.util.AssocList` specifies the following set definitions (using simple class names where possible):

```
usedSet fluid.util =
    AssocList, AssocKeyEnum,
    AssocValueEnum, Stack;
usedSet fluid.util.AssocList* =
    fluid.assoc.FixedLenAssocList,
    fluid.assoc.MonotonicAssocList;
```

The system can then use them to complete the following implicit annotations for the various access level modifiers:

```
private:  usedBy AssocList
default:  usedBy fluid.util
protected: usedBy fluid.util,
           fluid.util.AssocList*
public:   usedBy *
```

Here the `*` extension of a class name refers to the set of all its subclasses (not just its direct subclasses) and the `*` in the `usedBy` clause refers to the unbounded set of all classes. Without the explicit used set definitions, the implicit annotations are equivalent to `usedBy *`.

The `usedBy` annotations can be checked whenever a piece of client code refers to an entity in a different class, with an error indicated when the class of the client code is not in the used set. It is trivial for a tool to infer used sets if the whole program is available. Otherwise, the developer can make a large-scale promise (such as identifying all the classes that might use some class) which then could be used as part of an ongoing iterative process.

2.4 Summary

In this section, we have described a number of promises: effects promises enable safe reordering of code; unique promises enable safe insertion of destructive operations and of copying, and they make effects analysis more precise; “used by” promises allow escape from complete-program assumptions by limiting the scope of possible places in the code where a declaration can be used. Use of these promises can enable soundness guarantees for many kinds of program-restructuring activities that would otherwise be unsafe to carry out in larger systems and in distributed development efforts.

We suggest that promises such as these, unlike more complete functional specifications, can be directly managed by programmers working with analysis and manipulation tools (employing static analysis algorithms rather than general-purpose theorem proving), with tools taking a major role in identifying and validating promises, and tracking their use in manipulation. In those cases where a tool fails to validate a promise or assure a demand (due, for example, to lack of availability of a particular component), a dependency tracking mechanism can allow a software engineer can take responsibility for the promise, and resolve its status (and the status of changes which depend on it) at a later time.

3 MANAGING PROMISES

Promises offer a “currency of flexibility” for developers and maintainers of software, in the sense that the addition and removal of promises regulates the balance of flexibility between a component and its clients. When a component offers many promises, clients gain flexibility in the kinds of restructuring changes they can make. But an overly comprehensive set of promises can constrain the evolution of the component itself by limiting options for change within the component.

Thus, in general, the management of promises involves

decisions to add and remove promises (and demands), and techniques to facilitate the making and implementation of those decisions.

The `throws` clause in Java serves as a kind of promise, and its design is illustrative of more general issues of promise management. In Java, a method must declare the classes of exceptions that may be thrown during its execution. This set of thrown exceptions is computed from the types of exceptions used in `throw` statements and declared as thrown by methods called from that method. From this set are removed the exceptions caught by `catch` clauses. The remaining (uncaught) exceptions should be declared in the header of the method. This determination of which exceptions could be thrown out of a method body is necessarily conservative. For instance, suppose a method is declared as possibly throwing some exception, but it is only called under conditions in which no exception will be thrown. To a Java compiler, it will appear the potential exception is uncaught and should be declared in the `throws` clause of the calling method. Therefore, in order to keep the restriction from being too burdensome, those exceptions from the class `RuntimeException` and its subclasses need *not* be declared. This class covers common runtime errors such as dereferencing a null pointer or indexing an array outside of its bounds. When a developer defines an exception class, it can be made a subclass of `RuntimeException` to avoid the need to declare it in `throws` clauses, but then the benefit of the `throws` clause is lost. In order to avoid such dilemmas with promises, the developer can take formal responsibility (recorded by a tool) for promises that cannot yet be verified.

3.1 Offering promises

A naive analysis suggests that there are two possible approaches that can be followed in determining the correct set of promises to be offered at any point in the development or evolution of a system. These are building up and building down. The “building up” approach suggests that developers add promises, either reactively when the safety of a change to client code needs a particular promise to be made, or preemptively when there is high value in client flexibility at low cost in component flexibility. In this case, in order to offer the additional promise, the developer may need to extract promises regarding other components. Of course, the promises may not be immediately forthcoming, in which case the developer could choose to leave the promise unverified, and so record (using a dependency management tool) a potentially inconsistent dependency for later resolution.

The “building down” approach, on the other hand, suggests that developers use automatic tools to identify a large set of valid promises that can be offered relating to software component. This is possible for all three kinds

of promises described in Section 2, with programmer interaction needed only in specific instances, for example to define regions and identify unshared fields. In this case, the programmer may need to serve as a moderating force to assure that the balance of flexibility does not swing too far from component developer to component client as a result of an excessive weight of automatically generated promises. This is accomplished in two ways: prospectively by monitoring and managing the promises that are to be published for a component, and retrospectively by using tools to manage dependency information that identifies which promises actually contribute to assuring the soundness of code changes in client code. This dependency information can assist programmers in removing or weakening promises that have no dependencies.

In practice, extrinsic engineering considerations are likely to be the primary guide for decisions relating to this balance of flexibility. Such considerations would indicate, for example, which components are most likely to evolve over an anticipated life-cycle or across a product-line. The benefit of promises is that they offer a more precise and explicit way to manage this balance of flexibility, while assuring soundness of structural changes where they are most likely to be made. It remains an empirical issue to see what are the costs—how the balance can be effectively managed in practice, what are appropriate roles for tools, and how the implicit negotiation between component and clients can be supported.

3.2 Breaking promises

There are two ways in which a promise may be broken. It may turn out that a recorded but unvalidated promise cannot be kept. It may also happen that subsequent modifications to the code break the promise or violate a demand. In either case, it is important to find out where the promises have been used so that the extent of the problem can be determined.

4 RELATED WORK

Whole program analyses that identify properties such as aliasing and side-effects are too numerous to mention. Rather than helping the programmer to constrain the behavior of separately developed components (as in the case with promises), these analyses are usually used by sophisticated optimizing compilers. Sometimes, the compilers are assisted with promise-like annotations that make up for a lack of precision in the analysis by locally overriding the analysis results. Unlike promises, however, these annotations are not interprocedural in nature and they are not supported by a system that permits verification options ranging from fully proved to simply trusted.

Flanagan and Felleisen [8] describe a set-based dataflow analysis that integrates the results of separately analyzing components. As an aside, they propose summarizing the voluminous machine-generated set constraints with signatures that approximate the exact constraints. They speculate that these signatures could be programmer-specified.

Other researchers have used the technique of enhanced “header files” for breaking intractable global consistency issues into locally manageable statically checkable pieces. As noted above, LCLint [21, 6] used annotations on pointer types in C in order to statically check for dangling pointers and storage leakage. This project was carried out in the context of Larch [11], a formal specification and theorem proving system, which similarly used a variety of proof-supported resources for partially or fully specifying the behavior of separately developed entities. In particular, Larch supports mutability annotations. ‘Extended static checking’ [5, 15], which also uses a theorem prover, has been incorporated into experimental Modula-3 compilers. In particular, lock level annotations were successfully used in the Trestle window system to ensure the absence of deadlocks.

Jackson’s Aspect system [13] uses a similar abstraction mechanism to our regions in order to specify the effects of routines on abstract data types. He uses specifications for each procedure in order to allow checking to proceed in a modular fashion. In his work, however, the specifications are *necessary* effects, in order to check for missing functionality; whereas in our system the specified effects must include all effects of a method.

Effects were first studied in the FX system [9], a higher-order functional language with reference cells. The burden of manual specification of effects is lifted through the use of effects inference as studied by Gifford, Jouvelot, and Taplin [14, 20]. The work was motivated by a desire to use stack allocation instead of heap allocation in mostly pure functional programs, and also to assist parallel code generation. The approach was demonstrated successful for the former purpose in later work [22, 1]. These researchers also make use of a concept of disjoint “regions” of mutable state, but these regions are global, as opposed to within objects. In the original FX system, effects can be verified and exploited in separately developed program components.

Reynolds [18] originated the idea of “syntactic control of interference.” The intent was to develop an Algol-like programming language that used syntactic properties to eliminate “anonymous channels of interference.” Region promises are similar in intent, but differ in realization. They are selectively applied (to parts of a Java program) and allow specification of effects on parts of an object.

Adding notation for unique pointers was studied by

Minsky [16] in the context of Eiffel. Baker's 'use once' variables [4] show the fundamental rules that any such system should follow. Hogg [12] presented a notation for Smalltalk that enabled the identification of "islands," regions of objects accessible only through a single bridge point. This work has been made simpler and more powerful by Almeida [2], where it is shown that a range of types: balloon types, opaque balloon types, and value types can be declared and checked in the context of an Algol-like language. Objects within an island or balloon differ from our unshared fields in an object because aliases are permitted within an island. Furthermore, we permit an object to have both shared and unshared fields.

Our work differs from such language extensions because we do not rely on a single set of rules to verify promises. Promises additionally may be verified by external tools, or even by hand. This flexibility permits promises to be useful as soon as they are deployed rather than requiring an entire program to be annotated before it can be accepted.

Griswold and Notkin [10] have a tool to perform meaning-preserving restructuring on Scheme programs and describe a number of basic transformation steps. They keep the source, a program dependence graph (PDG) and control-flow graph (CFG) up-to-date with parallel transformations. The transformations are shown to preserve meaning using whole-program analysis. Their techniques would need to be modified to handle large or incomplete programs.

REFERENCES

- [1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM SIGPLAN PLDI'95 Conference*, pages 174–185, New York, June 1995. ACM Press.
- [2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming (11th European Conference)*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59, Berlin, June 1997. Springer-Verlag.
- [3] H. G. Baker. Lively linear Lisp — 'Look Ma, no garbage!'. *ACM SIGPLAN Notices*, 27(9):89–98, Aug. 1992.
- [4] H. G. Baker. 'Use-once' variables and linear objects: Storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, Jan. 1995.
- [5] D. L. Detlefs. An overview of the extended static checking system. In *First workshop on Formal Methods in Software Practice*, 1996.
- [6] D. Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN PLDI'96 Conference*, pages 44–53, New York, May 1996. ACM Press.
- [7] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: A tool for using specifications to check code. In *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 87–96, New Orleans, Louisiana, December 6–9 1994.
- [8] C. Flanagan and M. Felleisen. Componential set-based analysis. In *ACM SIGPLAN PLDI'97 Conference*, pages 235–248, Las Vegas, Nevada, June 15–18, 1997.
- [9] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, Sept. 1987.
- [10] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM TOSEM*, 2(3), 1993.
- [11] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, Berlin, Heidelberg, New York, 1993.
- [12] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA'91*, pages 271–285, New York, Nov. 1991. ACM Press.
- [13] D. Jackson. Aspect: Detecting bugs with abstract dependencies. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, Apr. 1995.
- [14] P. Jouvelot and D. K. Gifford. Algebraic reconstruction on types and effects. In *18th ACM POPL Conference*, pages 303–310. ACM Press, New York, Jan. 1991.
- [15] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, Pasadena, California, USA, 1995. Available as Technical Report Caltech-CS-TR-95-03.
- [16] N. Minsky. Towards alias-free pointers. In P. Cointe, editor, *ECOOP '96*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Berlin, Heidelberg, New York, July 1996. Springer-Verlag.
- [17] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.
- [18] J. C. Reynolds. Syntactic control of interference. In *5th Annual ACM POPL Symposium*, pages 39–46, January 23–25 1978.
- [19] W. L. Scherlis. Small-scale structural reengineering of software. In *ACM SIGSOFT 2nd International Workshop on Software Architecture*, pages 116–120. ACM Press, Oct. 1996.
- [20] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.
- [21] Y. M. Tan. Formal specification techniques for promoting software modularity, enhancing software documentation, and testing specifications. Technical Report MIT/LCS/TR-619, MIT, June 1994.
- [22] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *21st ACM POPL Conference*, pages 188–201, New York, Jan. 1994. ACM Press.

Gradual Typing for Objects

Jeremy Siek¹ and Walid Taha²

`jeremy.siek@colorado.edu, taha@rice.edu`

¹ University of Colorado, Boulder, CO 80309, USA

and LogicBlox Inc., Atlanta, GA 30309, USA

² Rice University, Houston, TX 77005, USA

Abstract. Static and dynamic type systems have well-known strengths and weaknesses. In previous work we developed a *gradual type system* for a functional calculus named $\lambda_{\rightarrow}^?$. Gradual typing provides the benefits of both static and dynamic checking in a single language by allowing the programmer to control whether a portion of the program is type checked at compile-time or run-time by adding or removing type annotations on variables. Several object-oriented scripting languages are preparing to add static checking. To support that work this paper develops $\mathbf{Ob}_{<:}^?$, a gradual type system for object-based languages, extending the $\mathbf{Ob}_{<:}$ calculus of Abadi and Cardelli. Our primary contribution is to show that gradual typing and subtyping are orthogonal and can be combined in a principled fashion. We also develop a small-step semantics, provide a machine-checked proof of type safety, and improve the space efficiency of higher-order casts.

1 Introduction

Static and dynamic typing have complementary strengths, making them better for different tasks and stages of development. Static typing provides full-coverage error detection, efficient execution, and machine-checked documentation whereas dynamic typing enables rapid development and fast adaptation to changing requirements. *Gradual typing* allows a programmer to mix static and dynamic checking in a program and provides a convenient way to control which parts of a program are statically checked. The goals for gradual typing are:

- Programmers may omit type annotations on parameters and immediately run the program; run-time type checks are performed to preserve type safety.
- Programmers may add type annotations to increase static checking. When all parameters are annotated, *all* type errors are caught at compile-time.³
- The type system and semantics should minimize the implementation burden on language implementors.

In previous work we introduced gradual typing in the context of a functional calculus named $\lambda_{\rightarrow}^?$ [47]. This calculus extends the simply typed lambda calculus

³ The language under study does not include arrays so the claim that we catch all type errors does not include the static detection of out-of-bound errors.

with a statically unknown (dynamic) type ? and replaces type equality with type consistency to allow for implicit coercions that add and remove ? s.

Developers of the object-oriented scripting languages Perl 6 [49] and JavaScript 4 [27] expressed interest in our work on gradual typing. In response, this paper develops the type theoretic foundation for gradual typing in object-oriented languages. Our work is based on the $\mathbf{Ob}^{<:}$ calculus of Abadi and Cardelli, a statically-typed object calculus with structural subtyping. We develop an extended calculus, named $\mathbf{Ob}_{<:}^?$, that adds the type ? and replaces the use of subtyping with a relation that integrates subtyping with type consistency.

The boundary between static and dynamic typing is a fertile area of research and the literature addresses many goals that are closely related to those we outline above. Section 8 describes the related work in detail.

The paper starts with a programmer’s and an implementor’s tour of gradual typing (Sections 2 and 3 respectively) before proceeding with the technical development of the new results in Sections 4 through 7.

Technical Contributions This paper includes the following original contributions:

1. The primary contribution of this paper shows that type consistency and subtyping are orthogonal and can be naturally superimposed (Section 4).
2. We develop a syntax-directed type system for $\mathbf{Ob}_{<:}^?$ (Section 5).
3. We define a semantics for $\mathbf{Ob}_{<:}^?$ via a translation to the intermediate language with explicit casts $\mathbf{Ob}_{<:}^{(\cdot)}$ for which we define a small-step operational semantics (Section 6).
4. We improve the space efficiency of the operational semantics for higher-order casts by applying casts in a lazy fashion to objects (Section 6).
5. We prove that $\mathbf{Ob}_{<:}^?$ is type safe (Section 7). The proof is a streamlined variant of Wright and Felleisen’s syntactic approach to type soundness [5, 53]. The formalization and proof are based on a proof of type safety for $\mathbf{FOb}_{<:}^?$ (a superset of $\mathbf{Ob}_{<:}^?$ that also includes functions) we wrote in the Isar proof language [52] and checked using the Isabelle proof assistant [39]. The formalization for $\mathbf{FOb}_{<:}^?$ is available in a technical report [46].
6. We prove that $\mathbf{Ob}_{<:}^?$ is statically type safe for fully annotated programs (Section 7), that is, we show that neither cast exceptions nor type errors may occur during program execution.

2 A Programmer’s View of Gradual Typing

We give a description of gradual typing from a programmer’s point of view, showing examples in hypothetical variant of the ECMAScript (aka JavaScript) programming language [15] that provides gradual typing. The following `Point` class definition has no type annotations on the data member `x` or the `dx` parameter. The gradual type system therefore delays checks concerning `x` and `dx` inside the `move` method until run-time, as would a dynamically typed language.

```

class Point {
    var x = 0
    function move(dx) { this.x = this.x + dx }
}
var a : int = 1
var p = new Point
p.move(a)

```

More precisely, because the types of the variables `x` and `dx` are statically unknown the gradual type system gives them the “dynamic” type, written `?` for short. The reader may wonder why we do not infer the type of `x` from its initializer `0`. We discuss the relation between gradual typing and type inference in Section 8. Now suppose the `+` operator expects arguments of type `int`. The gradual type system allows an *implicit coercion* from type `?` to `int`. This kind of coercion could fail (like a down cast) and therefore must be dynamically checked. In statically-typed object-oriented languages, such as Java and C#, implicit up-casts are allowed (they never fail) but not implicit down-casts. Allowing implicit coercions that may fail is *the* distinguishing feature of gradual typing and is what allows gradual typing to support dynamic typing.

To enable the gradual migration of code from dynamic to static checking, gradual typing allows for a mixture of the two and provides seamless interaction between them. In the example above, we define a variable `a` of type `int`, and invoke the dynamically typed `move` method. Here the gradual type system allows an implicit coercion from `int` to `?`. This is a safe coercion—it can never fail at run-time—however the run-time system needs to remember the type of the value so that it can check the type when it casts back to `int` inside of `move`.

Gradual typing also allows implicit coercions among more complicated types, such as object types. An object type is similar to a Java-style interface in that it contains a list of member signatures, however object types are compared structurally instead of by name. In the following example, the `equal` method has a parameter `o` annotated with the object type `[x:int]`.

```

class Point {
    var x = 0
    function bool equal(o : [x:int]) { return this.x == o.x }
}
var p = new Point
var q = new Point
p.equal(q)

```

The method invocation `p.equal(q)` is allowed by the gradual type system. The type of parameter `o` is `[x:int]` whereas the type of the argument `q` is `[x:?,equal:[x:int]→bool]`. We compare the two types structurally, one member at a time. For `x` we have a coercion from `?` to `int`, so that is allowed. Now consider the `equal` member. Because this is an object-oriented language with subtyping, we can use an object with more methods in a place that is expecting an object with fewer methods.

Next we look at a fully annotated program, that is, a program where all the variables are annotated with types. In this case the gradual type system acts like a static type system and catches *all* type errors during compilation. In the example below, the invocation of the annotated `move` method with a string argument is flagged as a static type error.

```
class Point {
    var x : int = 0
    function Point move(dx : int) { this.x = this.x + dx }
}
var p = new Point
p.move("hi") // static type error
```

3 An Implementor's View of Gradual Typing

Next we give an overview of gradual typing from a language implementor's point of view, describing the type system and semantics. The main idea of the type system is that we replace the use of type equality with type consistency, written \sim . The intuition behind type consistency is to check whether the two types are equal in the parts where both types are known. The following are a few examples. The notation $[l_1 : s_1, \dots, l_n : s_n]$ is an object type where $l : s$ is the name l and signature s of a method. A signature has the form $\tau \rightarrow \tau'$, where τ is the parameter type and τ' is the return type of the method.

$$\begin{array}{llll} \text{int} \sim \text{int} & \text{int} \not\sim \text{bool} & ? \sim \text{int} & \text{int} \sim ? \\ [x : \text{int} \rightarrow ?, y : ? \rightarrow \text{bool}] \sim [y : \text{bool} \rightarrow ?, x : ? \rightarrow \text{int}] \\ [x : \text{int} \rightarrow \text{int}, y : ? \rightarrow \text{bool}] \not\sim [x : \text{bool} \rightarrow \text{int}, y : ? \rightarrow \text{bool}] \\ [x : \text{int} \rightarrow \text{int}, y : ? \rightarrow ?] \not\sim [x : \text{int} \rightarrow \text{int}] \end{array}$$

To express the “where both types are known” part of the type consistency relation, we define a restriction operator, written $\sigma|_\tau$. This operator “masks off” the parts of type σ that are unknown in type τ . For example,

$$\begin{array}{ll} \text{int}|_? = ? & \text{int}|_{\text{bool}} = \text{int} \\ [x : \text{int} \rightarrow \text{int}, y : \text{int} \rightarrow \text{int}]|_{[x : ? \rightarrow ?, y : \text{int} \rightarrow \text{int}]} = [x : ? \rightarrow ?, y : \text{int} \rightarrow \text{int}] \end{array}$$

The restriction operator is defined as follows.

```
 $\sigma|_\tau = \text{case } (\sigma, \tau) \text{ of}$ 
 $(-, ?) \Rightarrow ?$ 
 $| ([l_1 : s_1, \dots, l_n : s_n], [l_1 : t_1, \dots, l_n : t_n]) \Rightarrow$ 
 $[l_1 : s_1|_{t_1}, \dots, l_n : s_n|_{t_n}]$ 
 $| (-, -) \Rightarrow \sigma$ 
```

```
 $(\sigma_1 \rightarrow \sigma_2)|_{(\tau_1 \rightarrow \tau_2)} = (\sigma_1|_{\tau_1}) \rightarrow (\sigma_2|_{\tau_2})$ 
```

Definition 1. Two types σ and τ are **consistent**, written $\sigma \sim \tau$, iff $\sigma|_\tau = \tau|_\sigma$, that is, when the types are equal where they are both known.⁴

Proposition 1. (Basic Properties of \sim)

1. \sim is reflexive.
2. \sim is symmetric.
3. \sim is not transitive. For example, $\text{bool} \sim ?$ and $? \sim \text{int}$ but $\text{bool} \not\sim \text{int}$.
4. $\tau \sim \tau|_\sigma$.
5. If neither σ nor τ contain $?$, then $\sigma \sim \tau$ iff $\sigma = \tau$.

A gradual type system uses type consistency where a simple type system uses type equality. For example, in the following hypothetical rule for method invocation, the argument and parameter types must be consistent.

$$\frac{\Gamma \vdash e_1 : [\dots, l : \sigma \rightarrow \tau, \dots] \quad \Gamma \vdash e_2 : \sigma' \quad \sigma' \sim \sigma}{\Gamma \vdash e_1.l(e_2) : \tau}$$

Gradual typing corresponds to static typing when no $?$ appear in the program (either explicitly or implicitly) because when neither σ nor τ contain $?$, we have $\sigma \sim \tau$ if and only if $\sigma = \tau$, as stated in Proposition 1.

Broadly speaking, there are two ways to implement the run-time behavior of a gradually typed language. One option is to erase the type annotations and interpret the program as if it were dynamically typed. This is an easy way to extend a dynamically typed language with gradual typing. The disadvantages of this approach is that unnecessary run-time type checks are performed and some errors become manifest later in the execution of the program. We do not describe this approach here as it is straightforward to implement.

The second approach performs run-time type checks at the boundaries of dynamically and statically typed code. The advantage is that statically typed code performs no run-time type checks. But there is an extra cost in that run-time tags contain complete types so that objects may be completely checked at boundaries. There are observable differences between the two approaches. The following example runs to completion with the first approach but produces an error with the second approach.

```
function unit foo(dx : int) { }
var x : ? = false; foo(x)
```

In this paper we give a high-level description of the second approach by defining a cast-inserting translation from $\mathbf{Ob}_{<}^?$ to an intermediate language with explicit casts named $\mathbf{Ob}_{<}^{(?)}$. The explicit casts have the form $\langle \tau \Leftarrow \sigma \rangle e$, where σ is the type of the expression e and τ is the target type. As an example of cast-insertion, consider the translation of the unannotated `move` method.

⁴ We chose the name “consistency” because it is analogous to the consistency of partial functions. This analogy can be made precise by viewing types as trees and then using the standard encoding of trees as partial functions from tree-paths to labels [41]. The $?$ s are interpreted as places where the partial function is undefined.

```

function move(dx) { this.x = this.x + dx }
~~> function ? move(dx : ?) { this.x = <?≤ int>(<int≤ ?>this.x + <int≤ ?>dx) }

```

We define the run-time behavior of $\mathbf{Ob}_{<}^{(·)}$ with a small-step operational semantics in Section 6. The operational semantics defines rewrite rules that simplify an expression until it is either a value or until it gets stuck (no rewrite rules apply). A stuck expression corresponds to an error. We distinguish between two kinds of errors: *cast errors* and *type errors*. A cast error occurs when the run-time type of a value is not consistent with the target type of the cast. Cast errors can be thought of as triggering exceptions, though for simplicity we do not model exceptions here. We categorize all other stuck expressions as type errors.

Definition 2. A program is **statically type safe** when neither cast nor type errors can occur during execution. A program is **type safe** when no type errors can occur during execution.

In Section 7 we show that any $\mathbf{Ob}_{<}^?$ program is *type safe* and that any $\mathbf{Ob}_{<}^?$ program that is fully annotated is *statically type safe*.

4 Combining Gradual Typing and Subtyping

In previous work we discovered that approaches to gradual typing based on subtyping and $?$ as “top” do not achieve *static type safety* for fully annotated terms [2]. The problem is that if you allow an implicit down-cast from “top” to any type $(? <: S)$, then you can use the normal up-cast rule $R <: ?$ and transitivity to deduce $R <: S$ for *any* two types R and S . The resulting type system therefore accepts all programs and does not reject programs that have static type errors. This discovery led us to the type consistency relation which formed the basis for our gradual type system for functional languages. However, subtyping is a central feature of object-oriented languages, so the question is how can we add subtyping to gradual type system while maintaining static type safety for fully annotated terms? It turns out to be as simple as adding subsumption:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

We do not treat $?$ as the top of the subtype hierarchy, but instead treat $?$ as neutral to subtyping, with only $? <: ?$. The following defines subtyping⁵

$$\text{int} <: \text{int} \quad \text{float} <: \text{float} \quad \text{bool} <: \text{bool} \quad ? <: ?$$

$$\text{int} <: \text{float} \quad [l_i : s_i]_{i \in 1 \dots n+m} <: [l_i : s_i]_{i \in 1 \dots n}$$

⁵ The calculus $\mathbf{Ob}_{<}^?$ does not include functions, so no subtyping rules for function types are provided here. The calculus $\mathbf{FOb}_{<}^?$ in the technical report [46] includes function types.

While the type system is straightforward to define, more care is needed to define 1) a type checking *algorithm* and 2) an operational semantics that takes subtyping into account. In this section we discuss the difficulties in defining a type checking algorithm and present a solution.

It is well known that a type checking algorithm cannot use the subsumption rule because it is inherently non-deterministic. (The algorithm would need to guess when to apply the rule and what target type to use.) Instead of using subsumption, the standard approach is to use the subtype relation in the other typing rules where necessary [41]. The following is the result of applying this transformation to our gradually typed method invocation rule.

$$\frac{\Gamma \vdash e_1 : [\dots, l : \sigma \rightarrow \tau, \dots] \quad \Gamma \vdash e_2 : \sigma' \quad \sigma' <: \sigma'' \quad \sigma'' \sim \sigma}{\Gamma \vdash e_1.l(e_2) : \tau}$$

This rule still contains some non-determinacy because of the type σ'' . We need a combined relation that directly compares σ' and σ .

Fortunately there is a natural way to define a relation that takes both type consistency and subtyping into account. To review, two types are consistent when they are equal where both are known, i.e., $\sigma \sim \tau$ iff $\sigma|_\tau = \tau|_\sigma$. To combine type consistency with subtyping, we replace type equality with subtyping.

Definition 3 (Consistent-Subtyping). $\sigma \lesssim \tau \equiv \sigma|_\tau <: \tau|_\sigma$

Here we apply the restriction operator to types σ and τ that may differ according to the subtype relationship, so we must update the definition of restriction to allow for objects of differing widths, as shown below.

$$\sigma|_\tau = \text{case } (\sigma, \tau) \text{ of}$$

$$(-, ?) \Rightarrow ?$$

$$| ([l_1 : s_1, \dots, l_n : s_n], [l_1 : t_1, \dots, l_m : t_m]) \text{ where } n \leq m \Rightarrow$$

$$[l_1 : s_1|_{t_1}, \dots, l_n : s_n|_{t_n}]$$

$$| ([l_1 : s_1, \dots, l_n : s_n], [l_1 : t_1, \dots, l_m : t_m]) \text{ where } n > m \Rightarrow$$

$$[l_1 : s_1|_{t_1}, \dots, l_m : s_m|_{t_m}, l_{m+1} : s_{m+1}, \dots, l_n : s_n]$$

$$| (-, -) \Rightarrow \sigma$$

$$(\sigma_1 \rightarrow \sigma_2)|_{(\tau_1 \rightarrow \tau_2)} = (\sigma_1|_{\tau_1}) \rightarrow (\sigma_2|_{\tau_2})$$

The following proposition allows us to replace the conjunction $\sigma' <: \sigma''$ and $\sigma'' \sim \sigma$ with $\sigma' \lesssim \sigma$ in the gradual method invocation rule.

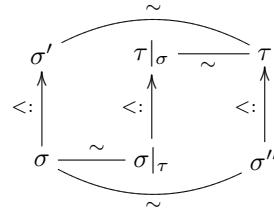
Proposition 2 (Properties of Consistent-Subtyping). *The following are equivalent:*

1. $\sigma \lesssim \tau$,
2. $\sigma <: \sigma'$ and $\sigma' \sim \tau$ for some σ' , and
3. $\sigma \sim \sigma''$ and $\sigma'' <: \tau$ for some σ'' .

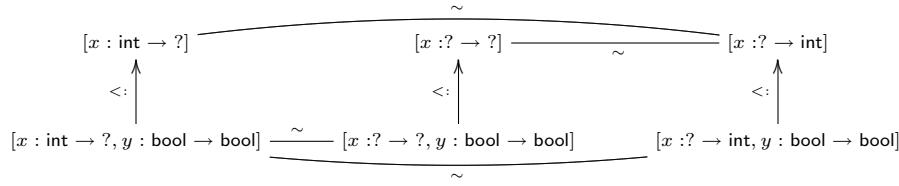
The method invocation rule can now be formulated in a syntax-directed fashion using the consistent-subtyping relation.

$$\frac{\Gamma \vdash e_1 : [\dots, l : \sigma \rightarrow \tau, \dots] \quad \Gamma \vdash e_2 : \sigma' \quad \sigma' \lesssim \sigma}{\Gamma \vdash e_1.l(e_2) : \tau}$$

It is helpful to think of the type consistency and subtyping relation as allowing types to differ along two different axes, with \sim along the x-axis and $<:$ along the y-axis. With this intuition, the following informal diagram represents Proposition 2.



The following is an example of the above diagram for a particular choice of types.



5 A Gradually Typed Object Calculus

We define a gradually typed object calculus named $\mathbf{Ob}_{<}^?$ by extending Abadi and Cardelli's $\mathbf{Ob}_{<}[\textcolor{red}{?}]$ with the unknown type $?$. For purposes of exposition, we add one parameter (in addition to `self`) to methods. The syntax of $\mathbf{Ob}_{<}^?$ includes three constructs for working with objects. The form $[l_i=\tau_i \varsigma(x_i : \sigma_i)e_i \ i \in 1 \dots n]$ creates an object containing a set of methods. Each method has a name l_i , a parameter x_i with type annotation σ_i , a body e_i , and a return type τ_i . The ς symbol just means “method” and is reminiscent of the λ used in functional calculi. The `self` parameter is implicit. Omitting a type annotation is short-hand for annotating with type $?$. Multi-parameter methods can be encoded using single-parameter methods $\textcolor{red}{[1]}$. The form $e_1.l(e_2)$ is a method invocation, where e_1 is the receiver object, l is the method to invoke, and e_2 is the argument. The form $e_1.l:=\tau \varsigma(x:\sigma)e_2$ is a method update. The result is a copy of e_1 except that its method l is replaced by the right-hand side. Abadi and Cardelli chose not to represent fields in the core calculus but instead encode fields as methods. The following is an example of a point object in $\mathbf{Ob}_{<}^?$:

`[equal=bool` $\varsigma(p:[x:int])$ `self.x.eq(p.x), x=zero]`.

Variables	$x \in \mathbb{X}$	$\supseteq \{\text{self}\}$	$e \in \mathbf{Ob}_{<:}$
Method labels	$l \in \mathbb{L}$		
Ground Types	$\gamma \in \mathbb{G}$	$\supseteq \{\text{bool, int, float, unit}\}$	
Constants	$c \in \mathbb{C}$	$\supseteq \{\text{true, false, zero, 0.0, ()}\}$	
Types	ρ, σ, τ	$::= \gamma \mid [l_i : s_i]_{i \in 1 \dots n}$	
Method Sig.	s, t	$::= \tau \rightarrow \tau$	
Expressions	e	$::= x \mid c \mid [l_i = \tau_i \varsigma(x_i : \sigma_i) e_i]_{i \in 1 \dots n} \mid e.l(e) \mid e.l := \tau \varsigma(x : \sigma)e$	
Syntactic Sugar	$l = e : \tau$	$\equiv l = \tau \varsigma(x : \text{unit})e \quad (x \notin e)$	
	$e.l$	$\equiv e.l(())$	
	$e_1.l := e_2 : \tau$	$\equiv e_1.l := \tau \varsigma(x : \text{unit})e_2 \quad (x \notin e)$	
Types	ρ, σ, τ	$+ = ?$	$e \in \mathbf{Ob}_{<:}^? \supset \mathbf{Ob}_{<:}$
Syntactic Sugar	$\varsigma(x)e$	$\equiv ? \varsigma(x : ?)e$	
	$l = e$	$\equiv l = e : ?$	
	$e_1.l := e_2$	$\equiv e_1.l := e_2 : ?$	

The gradual type system for $\mathbf{Ob}_{<:}^?$ is shown in Figure 1. (For reference, the type system for $\mathbf{Ob}_{<:}$ is in the Appendix, Fig. 4.) We use the symbol Γ for environments, which are finite partial functions from variables to types. The type system is parameterized on a *TypeOf* function that maps constants to types.

There are two rules for each elimination form. The first rule handles the case when the type of the receiver is unknown and the second rule handles when the type of the receiver is known. In the (GIVK1) rule for method invocation, the type of the receiver e_1 is unknown and the type of the argument e_2 is unconstrained. The rule (GIVK2) is described in Section 4, and is where we use the consistent-subtyping relation \lesssim . The rule (GUPD1) for method update handles the case when the type of the receiver e_1 is unknown. The new method body is type checked in an environment where `self` is bound to $?$ and the parameter x is bound to its declared type σ . The result type for this expression is $[l : \sigma \rightarrow \tau]$.⁶ The rule (GUPD2) handles the case for method update when the type of the receiver is an object type ρ . The new method body is type checked in an environment where `self` is bound to ρ and x is bound to its declared type σ . The constraints $\sigma_k \lesssim \sigma$ and $\tau \lesssim \tau_k$ make sure that the new method can be coerced to the type of the old method.

⁶ The result type for (GUPD1) is somewhat unsatisfactory because a method $l' \neq l$ can be invoked on e but not on the updated version of e . This can be easily resolved by extending the type system to include open object types in addition to closed object types, as is done in OCaml. If an object has an open object type you may invoke methods that are not listed in its type.

Fig. 1. A Gradual Type System for Objects.

$(GVAR)$	$\frac{\Gamma(x) = \tau}{\Gamma \vdash_G x : \tau}$	$\boxed{\Gamma \vdash_G e : \tau}$
$(GCONST)$	$\Gamma \vdash_G c : TypeOf(c)$	
$(GOBJ)$	$\frac{\Gamma, \mathbf{self} : \rho, x_i : \sigma_i \vdash_G e_i : \tau_i \quad \forall i \in 1 \dots n}{\begin{array}{c} \Gamma \vdash_G [l_i = \tau_i \varsigma(x_i : \sigma_i) e_i]_{i \in 1 \dots n} : \rho \\ (\text{where } \rho \equiv [l_i : \sigma_i \rightarrow \tau_i]_{i \in 1 \dots n}) \end{array}}$	
$(GIVK1)$	$\frac{\Gamma \vdash_G e_1 : ? \quad \Gamma \vdash_G e_2 : \tau}{\Gamma \vdash_G e_1.l(e_2) : ?}$	
$(GIVK2)$	$\frac{\Gamma \vdash_G e_1 : [\dots, l : \sigma \rightarrow \tau, \dots] \quad \Gamma \vdash_G e_2 : \sigma' \quad \sigma' \lesssim \sigma}{\Gamma \vdash_G e_1.l(e_2) : \tau}$	
$(GUPD1)$	$\frac{\Gamma \vdash_G e : ? \quad \Gamma, \mathbf{self} : ?, x : \sigma \vdash e' : \tau}{\Gamma \vdash_G e.l := \tau \varsigma(x : \sigma) e' : [l : \sigma \rightarrow \tau]}$	
$(GUPD2)$	$\frac{\Gamma \vdash_G e_1 : \rho \quad \Gamma, \mathbf{self} : \rho, x : \sigma \vdash_G e_2 : \tau \quad \sigma_k \lesssim \sigma \quad \tau \lesssim \tau_k}{\begin{array}{c} \Gamma \vdash_G e_1.l_k := \tau \varsigma(x : \sigma) e_2 : \rho \\ (\text{where } \rho \equiv [l_i : \sigma_i \rightarrow \tau_i]_{i \in 1 \dots n} \text{ and } k \in 1 \dots n) \end{array}}$	

6 A semantics for $\mathbf{Ob}_{<}^?$

In this section we define a semantics for $\mathbf{Ob}_{<}^?$ by defining a cast-inserting translation to the intermediate language $\mathbf{Ob}_{<}^{(?)}$ and by defining an operational semantics for $\mathbf{Ob}_{<}^{(?)}$. The syntax and typing rules for the intermediate language are those of $\mathbf{Ob}_{<}:$ [1] (Fig. 4 of the Appendix) extended with an explicit cast. The syntax and typing rule for the explicit cast are shown below.

Intermediate Language

$\text{Expressions } e \quad + = \langle \tau \Leftarrow \tau \rangle e$	$\boxed{e \in \mathbf{Ob}_{<}^{(?)} \supset \mathbf{Ob}_{<} :}$
\dots	$\frac{\Gamma \vdash e : \sigma \quad \sigma \sim \tau \quad \sigma \neq \tau}{\Gamma \vdash \langle \tau \Leftarrow \sigma \rangle e : \tau}$

Most run-time systems for dynamic languages associate a “type tag” with each value so that run-time type checks can be performed efficiently. In this paper we use a term-rewriting semantics that works directly on the syntax, without auxiliary structures. Instead of type tags, the cast expressions themselves are used to support run-time type checking. The cast includes both the source and

target type because both pieces of information are needed at run-time to apply casts to objects.

We do not allow “no-op” casts in the intermediate language to simplify the canonical forms of values, e.g., a value of type **int** is an integer, and not an integer cast to **int**. The typing rule for casts requires the source and target type to be consistent, so the explicit cast may only add or remove ?’s from the type. Implicit up-casts due to subtyping remain implicit using a subsumption rule, as such casts are safe and there is no need for run-time checking.

6.1 The Cast Insertion Translation

The cast insertion translation is guided by the gradual type system, inserting casts wherever the type of a subexpression differs from the expected type. For example, recall the rule for method invocation.

$$(GIVK2) \frac{\Gamma \vdash_G e_1 : [\dots, l : \tau \rightarrow \tau', \dots] \quad \Gamma \vdash_G e_2 : \sigma \quad \sigma \lesssim \tau}{\Gamma \vdash_G e_1.l(e_2) : \tau'}$$

The type σ of e_2 may differ from the method’s parameter type τ . We need to translate the invocation to a well typed term of $\mathbf{Ob}_{\lesssim}^{(<:)}$, where the argument type must be a subtype of the parameter type. We know that $\sigma \lesssim \tau$, so σ can differ from τ along both the type consistency relation \sim and the subtype relation $<:$. So we have the diagram on the left:



A cast can move us along the x-axis, and the subsumption rule can move us along the y-axis. So a solution to the problem, shown above on the right, is to cast e_2 from σ to some type ρ where $\rho <: \tau$. (We could just as well move up along the y-axis via subsumption before casting along the x-axis; it makes no difference.) The following example shows how we can choose ρ for a particular situation and gives some intuition for how we can choose it in general.

$$[x : \text{int} \rightarrow ?, y : \text{bool} \rightarrow \text{bool}] \xrightarrow{\sim} [x : ? \rightarrow \text{int}, y : \text{bool} \rightarrow \text{bool}] \xrightarrow{\lesssim} [x : ? \rightarrow \text{int}]$$

The type ρ must be the same width (have the same methods) as σ , and it must have a ? in all the locations that correspond to ?s in τ (and not have ?s where τ does not). In general, we can construct ρ with the merge operator, written $\sigma \leftarrow \tau$, defined below.

$$\sigma \leftarrow \tau \equiv \mathbf{case}(\sigma, \tau) \text{ of}$$

$$(_, _) \Rightarrow \tau$$

$$| (_, _) \Rightarrow ?$$

$$| ([l_1 : s_1, \dots, l_n : s_n], [l_1 : t_1, \dots, l_m : t_m]) \text{ where } n \leq m \Rightarrow$$

$$[l_1 : s_1 \leftarrow t_1, \dots, l_n : s_n \leftarrow t_n]$$

$$| ([l_1 : s_1, \dots, l_n : s_n], [l_1 : t_1, \dots, l_m : t_m]) \text{ where } n > m \Rightarrow$$

$$[l_1 : s_1 \leftarrow t_1, \dots, l_m : s_m \leftarrow t_m, l_{m+1} : s_{m+1}, \dots, l_n : s_n]$$

$$| (_, _) \Rightarrow \sigma$$

$$(\sigma_1 \rightarrow \sigma_2) \leftarrow (\tau_1 \rightarrow \tau_2) = (\sigma_1 \leftarrow \tau_1) \rightarrow (\sigma_2 \leftarrow \tau_2)$$

With the merge operator, we have the following diagram:

$$\begin{array}{ccc} & \nearrow \lesssim & \uparrow \lessdot \\ \sigma' & \xrightarrow{\gtrsim} & (\sigma' \leftarrow \sigma) \end{array}$$

Proposition 3 (Basic Properties of \leftarrow).

1. $(\sigma \leftarrow \sigma) = \sigma$
2. $\sigma \sim (\sigma \leftarrow \tau)$
3. If $\sigma \lesssim \tau$ then $(\sigma \leftarrow \tau) \lessdot \tau$.

The cast insertion judgment $\Gamma \vdash e \rightsquigarrow e' : \tau$ translates an expression e in the environment Γ to e' and determines that its type is τ . The cast insertion rule for method invocation (on known object types) is defined as follows using $\sigma' \leftarrow \sigma$ as the target of the cast on e_2 .

$$(\text{CIvk2}) \frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : [\dots, l : \sigma \rightarrow \tau, \dots] \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : \sigma' \quad \sigma' \lesssim \sigma}{\Gamma \vdash e_1.l(e_2) \rightsquigarrow e'_1.l(\langle\langle (\sigma' \leftarrow \sigma) \Leftarrow \sigma' \rangle\rangle e'_2) : \tau}$$

In the case when $\sigma' = \sigma$, we do not insert a cast, which is why we use the following helper function.

$$\langle\langle \tau \Leftarrow \sigma \rangle\rangle e \equiv \text{if } \sigma = \tau \text{ then } e \text{ else } \langle\langle \tau \Leftarrow \sigma \rangle\rangle e$$

The rest of the translation rules are straightforward. Fig. 2 gives the full definition of the cast insertion translation.

The cast-insertion judgment subsumes the gradual type system and additionally specifies how to produce the translation. In particular, a cast-insertion derivation can be created for precisely those terms accepted by the type system.

Proposition 4 (Cast Insertion and Gradual Typing).

$$\Gamma \vdash_G e : \tau \text{ iff } \exists e'. \Gamma \vdash e \rightsquigarrow e' : \tau.$$

When there is a cast insertion translation for term e , the resulting term e' is guaranteed to be a well-typed term of the intermediate language. Lemma 1 is used directly in the type safety theorem.

Fig. 2. Cast Insertion

		$\Gamma \vdash e \rightsquigarrow e' : \tau$
(CVAR)		$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \rightsquigarrow x : \tau}$
(GCONST)		$\Gamma \vdash c \rightsquigarrow c : TypeOf(c)$
(COBJ)		$\frac{\Gamma, \mathbf{self} : \rho, x_i : \sigma_i \vdash e_i \rightsquigarrow e'_i : \tau_i \quad \forall i \in 1 \dots n}{\Gamma \vdash [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i^{i \in 1 \dots n}] \rightsquigarrow [l_i = \tau_i \varsigma(x_i : \sigma_i)e'_i^{i \in 1 \dots n}] : \rho} \quad (\text{where } \rho \equiv [l_i : \sigma_i \rightarrow \tau_i]^{i \in 1 \dots n})$
(CIVK1)		$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : ? \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : \tau}{\Gamma \vdash_G e_1.l(e_2) \rightsquigarrow (\langle\langle l : \tau \rightarrow ? \rangle\rangle e'_1).l(e'_2) : ?}$
(CIVK2)		$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : [\dots, l : \sigma \rightarrow \tau, \dots] \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : \sigma' \quad \sigma' \lesssim \sigma}{\Gamma \vdash e_1.l(e_2) \rightsquigarrow e'_1.l(\langle\langle (\sigma' \leftarrow \sigma) \Leftarrow \sigma' \rangle\rangle e'_2) : \tau}$
(CUPD1)		$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : ? \quad \Gamma, \mathbf{self} : ?, x : \sigma \vdash e_2 \rightsquigarrow e'_2 : \tau}{\Gamma \vdash e_1.l := \tau \varsigma(x : \sigma)e_2 \rightsquigarrow (\langle\langle l : \sigma \rightarrow \tau \rangle\rangle e'_1).l := \tau \varsigma(x : \sigma)e'_2 : [l : \sigma \rightarrow \tau]}$
(CUPD2)		$\frac{\begin{array}{c} \Gamma \vdash e_1 \rightsquigarrow e'_1 : \rho \quad \Gamma, \mathbf{self} : \rho, x : \sigma \vdash e_2 \rightsquigarrow e'_2 : \tau \\ \sigma_k \lesssim \sigma \quad \tau \lesssim \tau_k \quad e_3 \equiv \langle\langle \tau_k \Leftarrow \tau \rangle\rangle [x \mapsto \langle\langle \sigma \Leftarrow \sigma_k \rangle\rangle y] e'_2 \end{array}}{\Gamma \vdash e_1.l_k := \tau \varsigma(x : \sigma)e_2 \rightsquigarrow e'_1.l_k := \tau_k \varsigma(y : \sigma_k)e_3 : \rho} \quad (\text{where } \rho \equiv [l_i : \sigma_i \rightarrow \tau_i]^{i \in 1 \dots n} \text{ and } k \in 1 \dots n)$

Lemma 1 (Cast Insertion is Sound).

If $\Gamma \vdash e \rightsquigarrow e' : \tau$ then $\Gamma \vdash e' : \tau$.

Proof. The proof is by induction on the cast insertion derivation. \square

The next lemma is needed to prove *static type safety*, that is, a fully annotated term is guaranteed to produce neither cast nor type errors. The set of fully annotated terms of $\mathbf{Ob}_{<}^?$ is exactly the $\mathbf{Ob}_{<}$ subset of $\mathbf{Ob}_{<}^?$. The function FV returns the set of variables that occur free in an expression.

Lemma 2 (Cast Insertion is the Identity for $\mathbf{Ob}_{<}$).

If $\Gamma \vdash e \rightsquigarrow e' : \tau$ and $e \in \mathbf{Ob}_{<}$ and $\forall x \in \text{FV}(e) \cap \text{dom}(\Gamma)$. $\Gamma(x) \in \mathbf{Ob}_{<}$ then $\Gamma \vdash e : \tau$ and $\tau \in \mathbf{Ob}_{<}$ and $e = e'$.

Proof. The proof is by induction on the cast insertion derivation. \square

Lemma 2 is also interesting for performance reasons. It shows that for fully annotated terms, no casts are inserted so there is no run-time type checking overhead.

6.2 Operational Semantics of $\text{Ob}^{(\cdot)}$

In this section we define a small-step, evaluation context semantics [17, 18, 53] for $\text{Ob}^{(\cdot)}$. Evaluation reduces expressions to values.

Definition 4 (Values and Contexts). Simple values are constants, variables, and objects. Values are simple values or a simple value enclosed in a single cast. An evaluation context is an expression with a hole in it (written \square) to mark where rewriting (reduction) may take place.

$$\begin{array}{ll} \text{Simple Values} & \xi ::= c \mid x \mid [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i]_{i \in 1 \dots n} \\ \text{Values} & v ::= \xi \mid \langle \tau \Leftarrow \tau \rangle \xi \\ \text{Contexts} & E ::= \square \mid E.l(e) \mid v.l(E) \mid E := \tau \varsigma(x : \tau)e \mid \langle \tau \Leftarrow \tau \rangle E \end{array}$$

The reduction rules are specified in Fig. 3. When a reduction rule applies to an expression, the expression is called a redex:

Definition 5 (Redex). $\text{redex } e \equiv \exists e'. e \longrightarrow e'$

The semantics is parameterized on a δ -function that defines the behavior of the primitive methods attached to the constants. The rule for method invocation (IVK) looks up the body of the appropriate method and substitutes the argument for the parameter. The primitive method invocation rule (DELTA) simply evaluates to the result of applying δ . In both the (IVK) and (DELTA) rules, the argument is required to be a value as indicated by the use of meta-variable v . Method update (UPD) creates a new object in which the specified method has been replaced.

The traditional approach to evaluating casts is to apply them in an eager fashion. For example, casting at function types creates a wrapper function with the appropriate casts on the input and output [19, 20, 21, 48].

$$\langle (\rho \rightarrow \nu) \Leftarrow (\sigma \rightarrow \tau) \rangle v \longrightarrow (\lambda x : \rho. \langle \nu \Leftarrow \tau \rangle (v (\langle \sigma \Leftarrow \rho \rangle x)))$$

The problem with this approach is that the wrapper functions can build up, one on top of another, using memory in proportion to the number of cast applications. The solution we use here is to delay the application of casts, and to collapse sequences of casts into a single cast. When a cast is applied to a value that is already wrapped in a cast, either the (MERGE) or (REMOVE) rule applies, or else the cast is a “bad cast”.

Definition 6 (Bad Cast).

$$\begin{aligned} \text{badcast } e &\equiv \exists v \rho \sigma \sigma' \tau. e = \langle \tau \Leftarrow \sigma' \rangle \langle \sigma \Leftarrow \rho \rangle v \wedge \rho \not\leq \tau \\ \text{BadCast } e &\equiv \exists E e'. e = E[e'] \wedge \text{badcast } e' \end{aligned}$$

The (MERGE) rule collapses two casts into a single cast, and is guarded by a type check. The target type of the resulting cast must be consistent with the inner source type ρ and it must be a subtype of the outer target type τ . We

Fig. 3. Reduction

(IVK)	$o.l_j(v) \longrightarrow [\text{self} \mapsto o, x_j \mapsto v]e_j$ (where $o \equiv [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i]_{i \in 1 \dots n}$)	$(1 \leq j \leq n)$	$e \longrightarrow e$
(DELTA)	$c.l(v) \longrightarrow \delta(c, l, v)$		
(UPD)	$\longrightarrow [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i]_{i \in \{1 \dots n\} - \{j\}}, l_j = \tau \varsigma(x : \sigma)e$	$(1 \leq j \leq n)$	
(MERGE)	$\frac{\rho \lesssim \tau \quad \rho \neq \tau}{\langle \tau \Leftarrow \sigma' \rangle \langle \sigma \Leftarrow \rho \rangle v \longrightarrow \langle \langle (\rho \leftarrow \tau) \Leftarrow \rho \rangle \rangle v}$		
(REMOVE)	$\frac{\rho = \tau}{\langle \tau \Leftarrow \sigma' \rangle \langle \sigma \Leftarrow \rho \rangle v \longrightarrow v}$		
(IVKCST)	$((\tau \Leftarrow \sigma)v_1).l_j(v_2) \longrightarrow (\tau_2 \Leftarrow \sigma_2)(v_1.l_j((\sigma_1 \Leftarrow \tau_1)v_2))$ (where $\sigma \equiv [\dots, l_j : \sigma_1 \rightarrow \sigma_2, \dots]$ and $\tau \equiv [\dots, l_j : \tau_1 \rightarrow \tau_2, \dots]$)		
(UPDCST)	$\longrightarrow \langle \tau \Leftarrow \sigma \rangle (v.l_j := \tau_2 \varsigma(z : \sigma_1) \langle \sigma_2 \Leftarrow \tau_2 \rangle [x \mapsto \langle \langle \tau_1 \Leftarrow \sigma_1 \rangle \rangle z]e)$ (where $\sigma \equiv [\dots, l_j : \sigma_1 \rightarrow \sigma_2, \dots]$ and $\tau \equiv [\dots, l_j : \tau_1 \rightarrow \tau_2, \dots]$)		
(STEP)	$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$		$e \longmapsto e$
(REFL)	$e \longmapsto^* e$		$e \longmapsto^* e$
(TRANS)	$\frac{e_1 \longmapsto^* e_2 \quad e_2 \longmapsto e_3}{e_1 \longmapsto^* e_3}$		

therefore use the \leftarrow operator and cast from ρ to $\rho \leftarrow \tau$. The (REMOVE) rule applies when the inner source and the outer target types are equal, and removes both casts.

The delayed action of casts on objects is “forced” when a method is invoked or updated. The rules (IVKCST) and (UPDCST) handle these cases.

7 Type Safety of $\mathbf{Ob}_{<}^?$

The bulk of this section is dedicated to proving that the intermediate language $\mathbf{Ob}_{<}^{(.)}$ is type safe. The type safety of our source language $\mathbf{Ob}_{<}^?$ is a consequence of the soundness of cast insertion and the type safety of the intermediate language. The type safety proof for the intermediate language has its origins in the syntactic type soundness approach of Wright and Felleisen [53], but is substan-

tially reorganized using some folklore.⁷ We begin with a top-down overview of the proof and then list the lemmas and theorems in the standard bottom-up fashion.

The goal is to show that if a term e_s is well-typed ($\vdash e_s : \tau$) and reduces in zero or more steps to e_f ($e_s \xrightarrow{*} e_f$), then $\vdash e_f : \tau$ and e_f is either a value or contains a bad cast or e_f can be further reduced. Note that the statement “ e_f is either a value or contains a bad cast or e_f can be further reduced” is equivalent to saying that e_f is not a *type error* as defined in Section 3. The proof of type safety is by induction on the reduction sequence. A reduction sequence (defined in Fig. 3) is either a zero-length sequence (so $e_s = e_f$), or a reduction sequence $e_s \xrightarrow{*} e_i$ to an intermediate term e_i followed by a reduction step $e_i \xrightarrow{} e_f$. In the zero-length case, where $e_s = e_f$, we need to show that if e_s is well-typed then it is not a type error. This is shown in the Progress Lemma. In the second case, the induction hypothesis tells us that e_i is well-typed. We then need to show that if e_i is well-typed and $e_i \xrightarrow{} e_f$ then e_f is well-typed. This is shown in the Preservation Lemma. Once we have a well-typed e_f , we can use the Progress Lemma to show that e_f is not a type error.

Progress Lemma Suppose that e is well-typed and not a value and does not contain a bad cast. We need to show that e can make progress, i.e., there is some e' such that $e \xrightarrow{} e'$. Therefore we need to show that e can be decomposed into an evaluation context E filled with a redex $e_1 (\exists e_2. e_1 \longrightarrow e_2)$ so that we can apply rule (STEP) to get $E[e_1] \xrightarrow{} E[e_2]$. The existence of such a decomposition is given by the Decomposition Lemma.⁸ In general, when the Progress Lemma fails for some language, it is because there is a mistake in the definition of evaluation contexts (which defines where evaluation should take place) or there is a mistake in the reduction rules, perhaps because a reduction rule is missing.

Preservation Lemma We need to show that if $\vdash e : \tau$ and $e \xrightarrow{} e'$ then $\vdash e' : \tau$. Because $e \xrightarrow{} e'$, we know there exists an E , e_1 , and e_2 such that $e = E[e_1]$, $e' = E[e_2]$, and $e_1 \longrightarrow e_2$. The proof consists of three parts, each of which is proved as a separate lemma.

1. From $\vdash E[e_1] : \tau$ we know that e_1 is well-typed ($\vdash e_1 : \sigma$) and the context E is well-typed. The typing judgment for contexts (defined in the Appendix,

⁷ The original proof of Wright and Felleisen requires the definition of faulty expressions which is more complicated than necessary because it relies on a proof by contradiction. Later type soundness proofs, such as [28, 38, 43], take a more direct approach. We use a proof organization similar to [5].

⁸ Our Decomposition Lemma differs from the usual Unique Decomposition Lemma (but is similar to Lemma A.15 in [5]) in that we include the premise that the expression is well-typed and conclude with a stronger statement than usual, that the hole is filled with a redex. The usual approach is to conclude with a hole filled with something, let us call it a *pre-redex*, that turns out to be either a redex or an ill-typed term. We do not prove uniqueness here because it is not necessary in the proof of type safety. Nevertheless, decompositions are unique for $\mathbf{Ob}_{\leq^{\langle\rangle}}$.

Fig. 5) assigns the context an input and output type, such as $\vdash E : \sigma \Rightarrow \tau$.
 (Subterm Typing)

2. Because e_1 is well-typed and $e_1 \rightarrow e_2$, e_2 is well-typed with the same type as e_1 . (Subject Reduction)
3. Filling E with e_2 produces an expression of type τ . More precisely, if $\vdash E : \sigma \Rightarrow \tau$ and $\vdash e_2 : \sigma$ then $\vdash E[e_2] : \tau$. (Replacement)

In general, Subterm Typing and Replacement hold for a language so long as evaluation contexts are properly defined. Subject Reduction, on the other hand, is highly dependent on the reduction rules of the language and is the crux of the type safety proof.

We now state the lemmas and theorems in the traditional bottom-up order, but without further commentary due to lack of space. We start with some basic properties of objects.

Proposition 5 (Properties of Objects).

1. If $\Gamma \vdash [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i^{i \in 1 \dots n}] : \rho$ where $\rho \equiv [l_i : \sigma_i \rightarrow \tau_i^{i \in 1 \dots n}]$ and $j \in 1 \dots n$ and $\Gamma, \text{self} : \rho, x_j : \sigma_j \vdash e' : \tau_j$ then $\Gamma \vdash [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i^{i \in \{1 \dots n\} - \{j\}}, l_j = \tau_j \varsigma(x_j : \sigma_j)e'] : \rho$.
2. If $[l_i : \sigma_i \rightarrow \tau_i^{i \in 1 \dots n}] <: [l_j : \rho_j \rightarrow \nu_j^{j \in 1 \dots m}]$ and $k \in 1 \dots m$ then $\rho_k = \sigma_k$ and $\nu_k = \tau_k$.

7.1 Progress

Towards proving the Progress Lemma, we show that values of certain types have canonical forms.

Lemma 3 (Canonical Forms).

1. If $\vdash v : \gamma$ then $\exists c \in \mathbb{C}. v = c$.
2. If $\vdash v : \rho$ where $\rho \equiv [l_i : \sigma_i \rightarrow \tau_i^{i \in 1 \dots n}]$ then $\exists \bar{x} \bar{e}. v = [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i^{i \in 1 \dots n}]$ or $\exists \bar{x} \bar{\sigma}. v = \langle \sigma \Leftarrow \rho \rangle [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i^{i \in 1 \dots n}]$.
3. $\nexists \xi : ?$ (simple values do not have type $?$)

The main work in proving Progress is proving the Decomposition Lemma.

Lemma 4 (Decomposition). If $\vdash e : \tau$ then $e \in \text{Values}$ or $\exists \sigma E e'. e = E[e'] \wedge (\text{redex } e' \vee \text{badcast } e')$.

Proof. By induction on the typing derivation using the Canonical Forms Lemma and Proposition 5. \square

Lemma 5 (Progress). If $\vdash e : \tau$ then $e \in \text{Values}$ or $\exists e'. e \mapsto e'$ or $\text{BadCast } e$.

Proof. Immediate from the Decomposition Lemma. \square

7.2 Preservation

Next we prove the Preservation Lemma and the three lemmas on which it relies: Subterm Typing, Subject Reduction, and Replacement.

Lemma 6 (Subterm Typing). *If $\vdash E[e] : \tau$ then $\exists \sigma. \vdash E : \sigma \Rightarrow \tau$ and $\vdash e : \sigma$.*

Proof. A straightforward induction on the typing derivation. \square

We assume that the δ function for evaluating primitives is sound.

Assumption 1 (δ -typability).

If $TypeOf(c) = [\dots, l : \sigma \rightarrow \tau, \dots]$ and $\vdash v : \sigma$ then $\vdash \delta(c, l, v) : \tau$.

Towards proving the Subject Reduction lemma, for the function application case we need the standard Substitution Lemma which in turn requires an Environment Weakening Lemma.

Definition 7. $\Gamma \subseteq \Gamma' \equiv \forall x\tau. \Gamma(x) = \tau$ implies $\Gamma'(x) = \tau$

Lemma 7 (Environment Weakening).

If $\Gamma \vdash e : \tau$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash e : \tau$.

Proof. A straightforward induction on the typing derivation. \square

Definition 8. We write $\Gamma \setminus \{x\}$ for Γ restricted to have domain $\text{dom}(\Gamma) \setminus \{x\}$.

Lemma 8 (Substitution).

If $\Gamma \vdash e_1 : \tau$ and $\Gamma(x) = \sigma$ and $\Gamma \setminus \{x\} \subseteq \Gamma'$ and $\Gamma' \vdash e_2 : \sigma$ then $\Gamma' \vdash [x \mapsto e_2]e_1 : \tau$.

Proof. By induction on the typing derivation. All cases are straightforward except for (OBJ) and (UPD) for which we use Environment Weakening. \square

Lemma 9 (Inversions on Typing Rules).

1. If $\Gamma \vdash c : \sigma \rightarrow \tau$ then there exists σ' and τ' such that $TypeOf(c) = \sigma' \rightarrow \tau'$ and $\sigma <: \sigma'$ and $\tau' <: \tau$.
2. If $\Gamma \vdash \langle \tau' \Leftarrow \sigma \rangle e : \tau$ then $\tau' <: \tau$ and $\sigma \sim \tau'$ and $\sigma \neq \tau'$ and $\Gamma \vdash e : \sigma$.
3. Suppose $\Gamma \vdash [l_i = \tau_i \varsigma (x_i : \sigma_i)e_i]_{i \in 1\dots n} : \tau$ and let $\rho \equiv [l_i : \sigma_i \rightarrow \tau_i]_{i \in 1\dots n}$. Then $\rho <: \tau$ and for any $j \in 1\dots n$ we have $\Gamma, \text{self} : \rho, x_j : \sigma_j \vdash e_j : \tau_j$.

Proof. The proofs are by induction on the typing derivation. \square

Lemma 10 (Subject Reduction). *If $\vdash e : \tau$ and $e \longrightarrow e'$ then $\vdash e' : \tau$.*

Proof. The proof is by induction on the typing derivation, followed by case analysis on the reduction.

(Inv) Use the Substitution and Inversion Lemmas and Proposition 5.

(Delta) Use δ -typability and the Inversion Lemma.

(Upd) Use Proposition 5 and the Inversion Lemma.

(Merge) Use Proposition 3 and the Inversion Lemma.

(Remove, InvCst, UpdCst) Use the Inversion Lemma. \square

Lemma 11 (Replacement). If $E : \sigma \Rightarrow \tau$ and $\vdash e : \sigma$ then $\vdash E[e] : \tau$.

Proof. A straightforward induction on the context typing derivation. \square

Lemma 12 (Preservation). If $e \mapsto e'$ and $\vdash e : \tau$ then $\vdash e' : \tau$.

Proof. Apply Subterm Typing to get a well-typed evaluation context and redex. Then apply Subject Reduction and Replacement. \square

7.3 Type Safety

Lemma 13 (Type Safety of $\mathbf{Ob}_{\leq}^{(?)}$). If $\vdash e : \tau$ and $e \mapsto^* e'$ then $\vdash e' : \tau$ and $e' \in \text{Values or BadCast } e'$ or $\exists e''. e' \mapsto e''$.

Proof. By induction on the evaluation steps. For the base case, where $e = e'$, we use Progress to show that e is either a value, a bad cast, or can make progress. For the case where $e_1 \mapsto^* e_2$ and $e_2 \mapsto e_3$, e_2 is well-typed by the induction hypothesis and therefore e_3 is well-typed by Preservation. Applying Progress to e_3 brings us to the conclusion. \square

Theorem 1 (Type Safety of $\mathbf{Ob}_{\leq}^{?}$). If $\vdash e_1 \rightsquigarrow e_2 : \tau$ and $e_2 \mapsto^* e_3$ then $\vdash e_3 : \tau$ and $e_3 \in \text{Values or BadCast } e_3$ or $\exists e_4. e_3 \mapsto e_4$.

Proof. The expression e_2 is well-typed because cast insertion is sound (Lemma 1). We then apply Lemma 13. \square

Theorem 2 (Static Type Safety of $\mathbf{Ob}_{\leq}^{?}$). If $e_1 \in \mathbf{Ob}_{\leq}$ and $\vdash e_1 \rightsquigarrow e_2 : \tau$ and $e_2 \mapsto^* e_3$ then $\vdash e_3 : \tau$ and $e_3 \in \text{Values or } \exists e_4. e_3 \mapsto e_4$.

Proof. By Lemma 2 we have $e_1 = e_2$, so e_2 does not contain any casts. By Lemma 13 we know that either e_3 is a value or a bad cast or can make progress. However, since e_2 did not contain any casts, there can be none in e_3 . \square

8 Related Work

Type Annotations for Dynamic Languages Several dynamic programming languages allow explicit type annotations, such as Common LISP [33], Dylan [16, 45], Cecil [10], Boo [13], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [36], the Bigloo [8, 44] dialect of Scheme [34], and the Strongtalk dialect of Smalltalk [6, 7]. In these languages, adding type annotations brings some static checking and/or improves performance, but the languages do not make the guarantee that annotating all parameters in the program prevents all type errors and type exceptions at run-time. This paper formalizes a type system that provides this stronger guarantee.

Soft Typing Static checking can be added to dynamically typed languages using static analyses. Cartwright and Fagan [9], Flanagan and Felleisen [22], Aiken, Wimmers, and Lakshman [3], and Henglein and Rehof [29, 30] developed analyses that can be used, for example, to catch bugs in Scheme programs [23, 30]. These analyses provide warnings to the programmer while still allowing the programmer to execute their program immediately (even programs with errors), thereby preserving the benefits of dynamic typing. However, the programmer does not control which portions of a program are statically checked: these whole-program analyses have non-local interactions. Also, the static analyses bear a significant implementation burden on developers of the language. On the other hand, they can be used to reduce the amount of run-time type checking in dynamically typed programs (Chambers et al. [11, 14]) and therefore could also be used to improve the performance of gradually typed programs.

Dynamic Typing in Statically Typed Languages Abadi et al. [2] extended a statically typed language with a *Dynamic* type and explicit injection (*dynamic*) and projection operations (*typecase*). Their approach does not satisfy our goals, as migrating code between dynamic and static checking not only requires changing type annotations on parameters, but also adding or removing injection and projection operations throughout the code. Our approach automates the latter.

Interoperability Gray, Findler, and Flatt [25] consider the problem of interoperability between Java and Scheme and extended Java with a *Dynamic* type with implicit casts. They did not provide an account of the type system, but their work provided inspiration for our work on gradual typing. Matthews and Findler [35] define an operational semantics for multi-language programs but require programmers to insert explicit “boundary” markers between the two languages, reminiscent of the explicit injection and projections of Abadi et al.

Tobin-Hochstadt and Felleisen [31] developed a system that provides convenient inter-language migration between dynamic and static languages on a per-module basis. In contrast, our goal is to allow migration at finer levels of granularity and to allow for partially typed code. Tobin-Hochstadt and Felleisen build *blame tracking* into their system and show that errors may not originate from statically typed modules. Our gradual type system enjoys a similar property. If all parameters in a term are annotated then no casts are inserted into the term during compilation provided the types of the free variables in the term do not mention ? (Lemma 2). Thus, no cast errors can originate from such a term.

Hybrid typing The Hybrid Type Checking of Flanagan et al. [21, 24] combines standard static typing with refinement types, where the refinements may express arbitrary predicates. The type system tries to satisfy the predicates using automated theorem proving, but when no conclusive answer is given, the system inserts run-time checks. This work is analogous to ours in that it combines a weaker and stronger type system, allowing implicit coercions between the two systems and inserting run-time checks. One notable difference between our sys-

tem and Flanagan’s is that his is based on subtyping whereas ours is based on type consistency.

Ou et al. [40] define a language that combines standard static typing with more powerful dependent typing. Implicit coercions are allowed to and from dependent types and run-time checks are inserted. This combination of a weaker and a stronger type system is again analogous to gradual typing.

Quasi-Static Typing Thatte’s Quasi-Static Typing [50] is close to our gradual type system but relies on subtyping and treats the unknown type as the top of the subtype hierarchy. In previous work [47] we showed that implicit down-casts combined with the transitivity of subtyping creates a fundamental problem that prevents the type system from catching all type errors even when all parameters in the program are annotated.

Riely and Hennessy [42] define a partial type system for $D\pi$, a distributed π -calculus. Their system allows some locations to be untyped and assigns such locations the type `lbad`. Their type system, like Quasi-Static Typing, relies on subtyping, however they treat `lbad` as “bottom”, which allows objects of type `lbad` to be implicitly coercible to any other type.

Gradual Typing The work of Anderson and Drossopoulou on BabyJ [4] is closest to our own. They develop a gradual type system for *nominal types* and their permissive type `*` is analogous to our unknown type `?`. Our work differs from theirs in that we address structural type systems.

Gronski, Knowles, Tomb, Freund, and Flanagan [26] provide gradual typing in the Sage language by including a `Dynamic` type and implicit down-casts. They use a modified form of subtyping to provide the implicit down-casts whereas we use the consistency relation. Their work does not include a result such as Theorem 2 of this paper which shows that all type errors are caught in programs with fully annotated parameters.

Herman alerted us to the space-efficiency problems in the traditional approach to higher-order casts. (We used the traditional approach in [47].) Concurrent to the work in this paper, Herman, Tomb, and Flanagan [31] proposed a solution a space-efficiency problem which, similar to our approach, delays the application of higher-order casts. However, the details of their approach are based on the coercion calculus from Henglein’s Dynamic Typing framework [29]. The coercion calculus can be viewed as a way to *compile* the explicit casts of this paper, removing the interpretive overhead of traversing types at run-time.

Type inference A language with gradual typing is syntactically similar to one with type inference [12, 32, 37]: both allow type annotations to be omitted. However, type inference does not provide the same benefits as dynamic typing (and therefore gradual typing). With type inference, programmers save the time it takes to write down the types but they must still go through the process of revising their program until the type inferencer accepts the program as well typed. As type systems are conservative in nature and of limited (though ever increasing) expressiveness, it may take some time to turn a program (even one

without any real errors) into a program to which the type inferencer can assign a type. The advantage of dynamic typing (and therefore of gradual typing) is that programmers may begin executing and testing their programs right away.

9 Conclusion and Future Work

The debate between dynamic and static typing has continued for several decades, with good reason. There are convincing arguments for both sides. Dynamic typing is better suited for prototyping, scripting, and gluing components, whereas static typing is better suited for algorithms, data-structures, and systems programming. It is common practice for programmers to start developing a program in a dynamic language and then translate to a static language later on. However, static and dynamic languages are often radically different, making this translation difficult and error prone. Ideally, migrating between dynamic to static could take place gradually and within one language.

In this paper we present the formal definition of an object calculus $\mathbf{Ob}_{<:}^?$, including its type system and operational semantics. This language captures the key ingredients for implementing gradual typing in object-oriented languages, showing how the type consistency relation can be naturally combined with subtyping. The calculus $\mathbf{Ob}_{<}^?$ provides the flexibility of dynamically typed languages when type annotations are omitted by the programmer and provides the benefits of static checking when all method parameters are annotated. The type system and run-time semantics of $\mathbf{Ob}_{<}^?$ are relatively straightforward, so it is suitable for practical languages.

As future work, we intend to investigate the interaction between gradual typing and Hindley-Milner inference [12, 32, 37], and we intend to apply static analyses (such as Soft Typing [9] or Henglein’s Dynamic Typing [29]) to reduce the number of run-time casts that must be inserted during compilation. There are a number of features we omitted from the formalization for the sake of keeping the presentation simple, such as recursive types and imperative update. We plan to add these features to our formalization in the near future. Finally, we intend to incorporate gradual typing into a mainstream dynamically typed programming language and perform studies to evaluate whether gradual typing can benefit programmer productivity.

Acknowledgments

We thank the anonymous reviewers for their suggestions. We thank Amer Diwan, Christoph Reichenbach, Ronald Garcia, Stephen Freund, David Herman, David Broman, and Cormac Flanagan for comments and feedback on drafts of this paper. This work was supported by NSF ITR-0113569 Putting Multi-Stage Annotations to Work, Texas ATP 003604-0032-2003 Advanced Languages Techniques for Device Drivers, and NSF SOD-0439017 Synthesizing Device Drivers.

Bibliography

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
- [2] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
- [3] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1994. ACM Press.
- [4] C. Anderson and S. Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82. Elsevier, 2003.
- [5] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. W. S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. May 2005.
- [6] G. Bracha. Pluggable type systems. In *OOPSLA '04 Workshop on Revival of Dynamic Languages*, 2004.
- [7] G. Bracha and D. Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.
- [8] Y. Bres, B. P. Serpette, and M. Serrano. Compiling scheme programs to .NET common intermediate language. In *2nd International Workshop on .NET Technologies*, Pilzen, Czech Republic, May 2004.
- [9] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.
- [10] C. Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.
- [11] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 49–70, New York, NY, USA, 1989. ACM Press.
- [12] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.
- [13] R. B. de Oliveira. The Boo programming language. <http://boo.codehaus.org>, 2005.
- [14] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 93–102, New York, NY, USA, 1995. ACM Press.
- [15] ECMA. *Standard ECMA-262: ECMAScript Language Specification*, 1999.
- [16] N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.
- [17] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine and the lambda-calculus. pages 193–217, 1986.
- [18] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
- [19] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming*, October 2002.
- [20] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming*, 2004.
- [21] C. Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.
- [22] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999.
- [23] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 23–32, New York, NY, USA, 1996. ACM Press.
- [24] C. Flanagan, S. N. Freund, and A. Tomb. Hybrid types, invariants, and refinements for imperative objects. In *FOOL/WOOD '06: International Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.

- [25] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.
- [26] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. Technical report, University of California, Santa Cruz, 2006.
- [27] E. T. W. Group. EcmaScript 4 netscape proposal.
- [28] C. A. Gunter, D. Remy, and J. G. Riecke. A generalization of exceptions and control in ml-like languages. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 12–23, New York, NY, USA, 1995. ACM Press.
- [29] F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.
- [30] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: Translating scheme to ml. In *FPCA '95, ACM SIGPLAN-SIGARCH Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California, June 1995.
- [31] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, April 2007.
- [32] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans AMS*, 146:29–60, 1969.
- [33] G. L. S. Jr. An overview of COMMON LISP. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107, New York, NY, USA, 1982. ACM Press.
- [34] R. Kelsey, W. Clinger, and J. R. (eds.). Revised⁵ report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.
- [35] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007.
- [36] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.
- [37] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
- [38] A. Nanevski. A modal calculus for exception handling. In *Intuitionistic Modal Logics and Applications Workshop (IMLA '05)*, Chicago, IL, June 2005.
- [39] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [40] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types (extended abstract). In *3rd IFIP International Conference on Theoretical Computer Science*, August 2004.
- [41] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
- [42] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–104, New York, NY, USA, 1999. ACM Press.
- [43] A. Sabry. Minml: Syntax, static semantics, dynamic semantics, and type safety. Course notes for b522, February 2002.
- [44] M. Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.
- [45] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.
- [46] J. Siek and W. Taha. Gradual typing for objects: Isabelle formaliztaion. Technical Report CU-CS-1021-06, University of Colorado, Boulder, CO, December 2006.
- [47] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.
- [48] W. Taha, H. Makholm, and J. Hughes. Tag elimination and jones-optimality. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 257–275, London, UK, 2001. Springer-Verlag.
- [49] A. Tang. Pugs blog.
- [50] S. Thatte. Quasi-static typing. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381, New York, NY, USA, 1990. ACM Press.
- [51] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.
- [52] M. Wenzel. *The Isabelle/Isar Reference Manual*. TU München, April 2004.
- [53] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Appendix

Fig. 4. The type system for $\mathbf{Ob}_{<:}$

(VAR)	$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$	$\boxed{\Gamma \vdash e : \tau}$
(CONST)	$\Gamma \vdash c : TypeOf(c)$	
(OBJ)	$\frac{\begin{array}{c} \Gamma, \mathbf{self} : \rho, x_i : \sigma_i \vdash e_i : \tau_i \quad \forall i \in 1 \dots n \\ \Gamma \vdash [l_i = \tau_i \varsigma(x_i : \sigma_i)e_i]_{i \in 1 \dots n} : \rho \end{array}}{\Gamma \vdash (l_i : \sigma_i \rightarrow \tau_i)_{i \in 1 \dots n} : \rho}$ (where $\rho \equiv [l_i : \sigma_i \rightarrow \tau_i]_{i \in 1 \dots n}$)	
(IVK)	$\frac{\Gamma \vdash e_1 : [\dots, l : \sigma \rightarrow \tau, \dots] \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1.l(e_2) : \tau}$	
(UPD)	$\frac{\begin{array}{c} \Gamma \vdash e_1 : \rho \quad \Gamma, \mathbf{self} : \rho, x : \sigma \vdash e_2 : \tau \quad \sigma_k <: \sigma \quad \tau <: \tau_k \\ \Gamma \vdash e_1.l_k := \tau \varsigma(x : \sigma)e_2 : \rho \end{array}}{\Gamma \vdash e_1.l_k := \tau \varsigma(x : \sigma)e_2 : \rho}$ (where $\rho \equiv [l_i : \sigma_i \rightarrow \tau_i]_{i \in 1 \dots n}$ and $k \in 1 \dots n$)	
(SUB)	$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau}$	

Fig. 5. Well-typed contexts.

(CxHOLE)	$\vdash [] : \tau \Rightarrow \tau$	$\boxed{\vdash E : \tau \Rightarrow \tau}$
(CxIvkL)	$\frac{\vdash E : \sigma \Rightarrow [\dots, l : \rho \rightarrow \tau, \dots] \quad \vdash e : \rho}{\vdash E.l(e) : \sigma \Rightarrow \tau}$	
(CxIvkR)	$\frac{\vdash e : [\dots, l : \rho \rightarrow \tau, \dots] \quad \vdash E : \sigma \Rightarrow \rho}{\vdash e.l(E) : \sigma \Rightarrow \tau}$	
(CxUpd)	$\frac{\begin{array}{c} \vdash E : \sigma' \Rightarrow \rho \quad \mathbf{self} : \rho, x : \sigma \vdash e : \tau \quad \sigma_k <: \sigma \quad \tau <: \tau_k \\ \vdash E.l_k := \tau \varsigma(x : \sigma)e : \sigma' \Rightarrow \rho \end{array}}{\vdash E.l_k := \tau \varsigma(x : \sigma)e : \sigma' \Rightarrow \rho}$ (where $\rho \equiv [l_i : \sigma_i \rightarrow \tau_i]_{i \in 1 \dots n}$ and $1 \leq k \leq n$)	
(CxSub)	$\frac{\vdash E : \sigma \Rightarrow \rho \quad \vdash \rho <: \rho'}{\vdash E : \sigma \Rightarrow \rho'}$	

Capabilities for Uniqueness and Borrowing*

Philipp Haller and Martin Odersky

EPFL, Switzerland
`{philipp.haller, martin.odersky}@epfl.ch`

Abstract. An important application of unique object references is safe and efficient message passing in concurrent object-oriented programming. However, to prevent the ill effects of aliasing, practical systems often severely restrict the shape of messages passed by reference. Moreover, the problematic interplay between destructive reads—often used to implement unique references—and temporary aliasing through “borrowed” references is exacerbated in a concurrent setting, increasing the potential for unpredictable run-time errors.

This paper introduces a new approach to uniqueness. The idea is to use capabilities for enforcing both at-most-once consumption of unique references, and a flexible notion of uniqueness. The main novelty of our approach is a model of uniqueness and borrowing based on simple, unstructured capabilities. The advantages are: first, it provides simple foundations for uniqueness and borrowing. Second, it can be formalized using a relatively simple type system, for which we provide a complete soundness proof. Third, it avoids common problems involving borrowing and destructive reads, since unique references subsume borrowed references.

We have implemented our type system as an extension to Scala. Practical experience suggests that our system allows type checking real-world actor-based concurrent programs with only a small number of additional type annotations.

1 Introduction

Message-based concurrency provides robust programming models that scale from multi-core processors to distributed systems, web applications, and cloud computing. Seamless scalability requires that local and remote message send operations should behave the same. A good candidate for such a uniform semantics is that a sent message gets moved from the memory region of the sender to the (possibly disjoint) memory region of the receiver. Thus, a message is no longer accessible to its sender after it has been sent. This semantics also avoids data races if concurrent processes running on the same computer communicate only by passing messages.

However, moving messages physically requires expensive marshaling (i.e., copying). This would prohibit the use of message-passing altogether in performance-critical code that deals with large messages, such as image processing pipelines or network protocol stacks [19, 20]. To achieve the necessary performance in these applications, the underlying implementation must pass messages between processes running on the

* Final version published at the 24th European Conference on Object-Oriented Programming (ECOOP’10), June 2010, Maribor, Slovenia.

same shared-memory computer by reference. But reference passing makes it challenging to enforce race freedom, especially in the context of imperative, object-oriented languages, where aliasing is common. The two main approaches to address this problem are:

- Immutable messages. Only allow passing objects of immutable type. Examples are Java-style primitive types (e.g., `int`, `boolean`), immutable strings, and tree-shaped data, such as XML.
- Alias-free messages. Only a single, unique reference may point to each message; upon transfer, the unique reference becomes unusable [20, 39, 40].

Immutable messages are used, for instance, in Erlang [3], a programming language created by Ericsson that was used at first in telecommunication systems, but is now also finding applications in Internet commerce (e.g., Amazon’s SimpleDB¹).

The second approach usually imposes constraints on the shape of messages (e.g., trees [40]). Even though messages are passed by reference, message shape constraints may lead indirectly to copying overheads; data stored in an object graph that does not satisfy the shape constraints must first be serialized into a permitted form before it can be sent within a message.

Scala [35] provides Erlang-style concurrent processes as part of its standard library in the actors package [25]. Scala’s actors run on the standard Java platform [31]; they are gaining rapidly support in industry, with applications in the Kestrel message queue system² powering the popular Twitter micro-blogging service, and others.

In Scala actors, messages can be any kind of data, mutable as well as immutable. When sending messages between actors operating on the same computer, the message state is not copied; instead, messages are transferred by reference only. This makes the system flexible and guarantees high performance. However, race safety has previously neither been enforced by the language, nor by the run-time library.

This paper proposes a new type-based approach to statically enforce race safety in Scala’s actors. Our main goal is to ensure race safety with a type system that’s simple and expressive enough to be deployed in production systems by normal users. Our system removes important limitations of existing approaches concerning permitted message shapes. At the same time it allows interesting programming idioms to be expressed with fewer annotations than previous work, while providing equally strong safety guarantees.

1.1 Background

There exists a large number of proposals for unique object references. A comprehensive survey is beyond the scope of this paper; Clarke and Wrigstad [12] provide a good overview of earlier work, where unique references are not allowed to point to internally-aliased objects, such as doubly-linked lists. Aliases that are strictly internal to a unique object are not observable by external clients and are therefore harmless [48]. Importantly, “external” uniqueness enables many interesting programming patterns, such as

¹ See <http://aws.amazon.com/simpledb/>.

² See <http://github.com/robey/kestrel/>.

Proposal	Type System	Unique Objects	Encapsulation	Program Annotations
Islands	(~ linear types)	alias-free	full	type qualifiers, purity
Balloons	(abstr. interpr.)	alias-free	full	type qualifiers
PacLang	quasi-linear types	alias-free, flds. prim.	full	type qualifiers
PRFJ	expl. ownership	alias-free	deep/full	owners, regions, effects
StreamFlex	impl. ownership	alias-free, flds. prim.	full	type qualifiers
Kilim	impl. ownership	alias-free	full	type qualifiers
External U.	expl. ownership	intern. aliases	deep	owners, borrowing
UTT	impl. ownership	intern. aliases	deep	type qualifiers, regions
BR	capabilities	intern. aliases	deep	type qual., regions, effects
MOAO	expl. ownership	intern. aliases	full	simple owners, borrowing
Sing#	capabilities	intern. aliases	full	type qualifiers, borrowing
This paper	capabilities	intern. aliases	full	type qualifiers

Fig. 1. Proposals for uniqueness with (a) full encapsulation, (b) internal aliases, or (c) both

merging of data structures and abstraction of object creation (through factory methods [23]). In the following we consider two kinds of alias encapsulation policies:

- *Deep encapsulation*: [33] the only access (transitively) to the internal state of an object is through a single entry point. References to external state are allowed.
- *Full encapsulation*: same as deep encapsulation, except that no references to objects outside the encapsulated object from within the encapsulation boundary are permitted.

Our motivation to study full encapsulation is concurrent programming, where deep encapsulation is generally not sufficient to avoid data races. Figure 1 compares proposals from the literature that provide either uniqueness with internal aliasing, full alias encapsulation, or both. (Section 7 discusses other related work on linear types, regions, and program logics.) We classify existing approaches according to (a) the kind of type system they use, (b) the notion of unique/linear objects they support, (c) the alias encapsulation they provide, and (d) the program annotations they require for static (type) checking. We distinguish three main kinds of type systems: explicit (parametrized) ownership types [14], implicit ownership types, and systems based on capabilities/permissions. The third column specifies whether unique objects are allowed to have internal aliases; in general, alias-free unique references may only point to tree-shaped object graphs. The fourth column indicates the encapsulation policy. We explain the annotations in the fifth column in the context of each proposal.

Islands [28] provide fully-encapsulated objects protected by “bridge” classes. However, extending an Island requires unique objects, which must be alias-free. Almeida’s Balloon Types [1] provide unique objects with full encapsulation; however, the unique object itself may not be (internally) aliased. Ennals et al. [19] have used quasi-linear types [30] for efficient network packet processing in PacLang; in their system, packets may not contain nested pointers. The PRFJ language of Boyapati et al. [8] associates owners with shared-memory locks to verify correct lock acquisition. PRFJ does not support unique references with internal aliasing; it requires adding explicit owner parameters to classes and read/write effect annotations. StreamFlex [39] (like its suc-

cessor Flexotasks [4]) supports stream-based programming in Java. It allows zero-copy message passing of “capsule” objects along linear filter pipelines. Capsule classes must satisfy stringent constraints: their fields may only store primitive types or arrays of primitive types. Kilim [40] combines type qualifiers with an intra-procedural shape analysis to ensure isolation of Java-based actors. To simplify the alias analysis and annotation system, messages must be tree-shaped. StreamFlex, Flexotasks, and Kilim are systems where object ownership is enforced implicitly, i.e., types in their languages do not have explicit owners or owner parameters. This keeps their annotation systems pleasingly simple, but significantly reduces expressivity: unique objects may not be internally-aliased. Universe Types [17, 16] is a more general implicit ownership type system that restricts only object mutations, while permitting arbitrary aliasing. Universe Types are particularly attractive for us, because its type qualifiers are very lightweight. In fact, some of the annotations proposed in this paper are very similar, suggesting a close connection. Generally, however, the systems are very different, since restricting only modifications of objects does not prevent data races in a concurrent setting. UTT [32] extends Universe Types with ownership transfer; it increases the flexibility of external uniqueness by introducing explicit regions (“clusters”); an additional static analysis helps avoiding common problems of destructive reads. In Vault [21] Fähndrich and De-Line introduce adoption and focus for embedding linear values into aliased containers (adoption), providing a way to recover linear access to such values (focus). Their system builds on Alias Types [45] that allow a precise description of the shape of recursive data structures in a type system. Boyland and Retert [9] (BR in Figure 1) generalize adoption to model both effects and uniqueness. While their type language is very expressive, it is also clearly more complex than Vault. Their realized source-level annotations include region (“data group”) and effect declarations. MOAO [13] combines a minimal notion of ownership, external uniqueness, and immutability into a system that provides race freedom for active objects [51, 10]. To reduce the annotation burden messages have a flat ownership structure: all objects in a message graph have the same owner. It requires only simple owner annotations; however, borrowing requires existential owners [49] and owner-polymorphic methods. Sing# [20] uses capabilities [21] to track the linear transfer of message records that are explicitly allocated in a special exchange heap reserved for inter-process communication. Their tracked pointers may have internal aliases; however, storing a tracked pointer in the heap requires dynamic checks that may lead to deadlocks. Their annotation system consists of type qualifiers as well as borrowing (“expose”) blocks for accessing fields of unique objects.

Summary. In previous proposals, borrowing has largely been treated as a second-class citizen. Several researchers [9, 32] have pointed out the problems of ad-hoc type rules for borrowing (particularly in the context of destructive reads). Concurrency is likely to exacerbate these problems. However, principled treatments of borrowing currently demand a high toll: they require either existential ownership types with owner-polymorphic methods, or type systems with explicit regions, such as Universe Types with Transfer or Boyland and Retert’s generalized adoption. Both alternatives significantly increase the syntactic overhead and are extremely challenging to integrate into practical object-oriented programming languages.

Contribution. We introduce a type system that uses capabilities for enforcing both a flexible notion of uniqueness and at-most-once consumption of unique references, making the system uniform and simple. Our approach identifies uniqueness and borrowing as much as possible. In fact, the only difference between a unique and a borrowed object is that the unique object comes with the capability to consume it (e.g., through ownership transfer). While uniform treatments of uniqueness and borrowing exist [21, 9], our approach requires only simple, unstructured capabilities. This has several advantages: first, it provides simple foundations for uniqueness and borrowing. Second, it requires neither existential ownership nor explicit region declarations in the type system. Third, it avoids the problematic interplay between borrowing and destructive reads, since unique references subsume borrowed references. This paper also contributes:

- *A simple and flexible annotation system.* We introduce a small number of source-level annotations used to guide the type checker (Section 2). The system is simple in the sense that only local variables, fields and method parameters are annotated. This means that type declarations remain unchanged. This facilitates the integration of our annotation system into full-featured languages, such as Scala.
- *A simple formal model with soundness proof.* We formalize our type system in the context of an imperative object calculus (Section 3) and prove it sound (Section 4). (A complete proof of soundness appears in the companion technical report [24].) Our main point of innovation is a novel way to support internal aliasing of unique references, which is surprisingly simple. By protecting all aliases pointing into a unique object (graph) with the same capability, illegal aliases are avoided by consuming that capability. The formal model corresponds closely to the annotation system described in Section 2: all types in the formalization can be expressed using those annotations.
- *A practical design.* We extend our system to support closures and nested classes (Section 5), features that, so far, have been almost completely ignored by existing work on unique object references. However, we found these features to be indispensable for type-checking real-world Scala code, such as collection classes. We have implemented our type system as a pluggable annotation checker for the EPFL Scala compiler (Section 6). We show that real-world actor-based concurrent programs can be type-checked with only a small increase in type annotations.

2 Overview

As a running example we use the simplified core of partest, the parallel testing framework used to test the Scala compiler and standard libraries. The framework uses actors—lightweight concurrent processes—for running multiple tests in parallel, thereby achieving significant speed-ups on multi-core processors.

In this application, a master actor creates multiple worker actors, each of which receives a list of tests to be run. A worker executes the `runTests` method shown in Figure 2, which prompts the test execution. Each test is associated with a log file that records the output produced by compiling and, in some cases, running the test. These log files are collected in the `logs` list that the worker sends back to the master upon completing the test execution.

```

def runTests(kind: String, tests: List[Files]) {
  var succ, fail = 0
  val logs: LogList @unique = new LogList
  for (test <- tests) {
    val log:LogFile @unique = createLogFile(test)
    // run test...
    logs.add(log)
  }
  report(succ, fail, logs)
}
def report(succ: Int, fail: Int, logs: LogList @unique) {
  master ! new Results(succ, fail, logs)
}

```

Fig. 2. Running tests and reporting results.

Note that log files are neither immutable nor cloneable.³ Therefore, it is impossible to create a copy of the log files upon sending them to the master. To ensure that passing `logs` by reference is safe, we annotate its type as `@unique`. Inside the for-comprehension, we also annotate the `log` variable, which refers to a single log file, as `@unique`; this enables adding `log` to `logs` without losing the uniqueness of `logs`. (Below we explain how to check that the invocation of `add` is safe.)

The worker reports the test results to its master using `report`. The `@unique` annotation requires the `logs` parameter to be unique. Moreover, it indicates that the caller loses the permission to access the passed argument subsequently. In fact, any object reachable from `logs` becomes inaccessible to the caller. Conversely, `report` has the full permission to access `logs`. This allows sending it as part of a unique `Results` message to the master. Sending a unique object (using the `!` method) makes it unusable, as well as all objects reachable from it, including the `logs`.

In the above example we have shown how to use the `@unique` annotation to ensure the safety of passing message objects by reference. In the following we introduce aliasing invariants of our type system that guarantee the soundness of this approach.

Alias Invariant. The alias invariant that our system guarantees is based on a separation predicate on stack variables. (Below, we extend this invariant to fields.) We characterize two variables x, y as being *separate*, written $\text{separate}(x, y)$, if and only if they do not share a common reachable object.⁴ In other words, two variables are separate if they point to disjoint object graphs in the heap. Based on this predicate we define what it means for a variable to be *separately-unique*.

Definition 1 (Separate Uniqueness) A variable x is *separately-unique* iff $\forall y \neq x. y \text{ accessible} \Rightarrow \text{separate}(x, y)$.

³ `LogFile` inherits from `java.io.File`, which is not cloneable.

⁴ For simplicity we leave the heap implicit in the following discussion; we formalize it precisely in Section 3.



Fig. 3. Comparing (a) external uniqueness and (b) separate uniqueness. (\Rightarrow unique reference, \rightarrow legal reference, $--\rightarrow$ illegal reference)

(A variable is accessible if it may be accessed in the current program execution.) This definition of uniqueness implies that if x is a separately-unique variable, there is no other accessible variable on the stack that shares a common reachable object with x .

In contrast, this does not hold for external uniqueness [12], which is the notion of uniqueness most closely related to ours. Figure 3 compares the two notions of uniqueness. We assume that object A owns object B. This means that references r and i are internal to the ownership context of A . Ownership makes reference f' illegal. u is a unique reference to A ; uniqueness makes reference f illegal. Importantly, external uniqueness permits the s reference, which points to an object that is reachable without using u . Therefore, even if u is unusable, the target of s is still reachable. In contrast, our system enforces full encapsulation by forbidding the s reference. This means that making u unusable results in all objects reachable using u being unusable. Therefore, separate uniqueness avoids races when unique references are passed among concurrent processes (we prove this in the companion technical report [24]). With external uniqueness, one has to enforce additional constraints to ensure safety [13].

We are now ready to state the alias invariant that our type system provides.

Definition 2 (Alias Invariant) *Unique parameters are separately-unique.*

Note that this invariant does not require unique *variables* to be separately-unique. In particular, unique variables may be aliased by local variables on the stack. However, it is valid to pass a unique variable to a method expecting a unique argument. This means that it must always be possible to make unique variables separately-unique. In the following we explain how we can enforce this using a system of capabilities.

Capabilities. A unique variable has a type guarded by some capability ρ , written $\rho \triangleright T$ (typically, T is the underlying class type). Capabilities have two roles: first, they serve as static names for (disjoint) regions of the heap. Second, they embody access permissions [44, 9, 11] to those regions. The typing rules of our system consume and produce sets of capabilities. A variable with a type guarded by ρ can only be accessed if ρ is available, i.e., if it is contained in the input set of capabilities in the typing rule. Therefore, consuming ρ makes all variables of types guarded by ρ unusable. The following invariant expresses the fact that accessible variables guarded by different capabilities point to disjoint object graphs.

Definition 3 (Capability Type Invariant) Let x be a unique variable with guarded type $\rho \triangleright T$. If y is an accessible variable such that $\neg\text{separate}(x, y)$, then y has guarded type $\rho \triangleright S$.

Note that the above definition permits variables z of guarded type $\delta \triangleright U$ ($\delta \neq \rho$) such that $\neg\text{separate}(x, z)$. This is safe as long as δ is not available, which makes z inaccessible.

In summary, the above invariant implies that if x 's type is guarded by some capability ρ , consuming ρ makes all variables y such that $\neg\text{separate}(x, y)$ inaccessible. Therefore, the separate uniqueness of unique arguments can be enforced as follows: first, unique arguments must have guarded type $\rho \triangleright T$. Second, capability ρ is consumed (and, therefore, must be available) in the caller's context. Third, capabilities guarding other arguments (if any) must be different from ρ . In Section 3 we formalize the mapping between annotations in the surface syntax, such as `@unique`, and method types with capabilities.

We have introduced two invariants that are fundamental to the soundness of unique variables and parameters in our system. In the following we continue the discussion of our running example, thereby motivating extensions of our annotation system.

Transient and peer parameters. Our discussion of the example shown in Figure 2 did not address the problem of mutating the unique `logs` list after running a single test. Crucially, `logs` must remain unique (and accessible) after adding `log` to it. This means we cannot use `@unique` to annotate the receiver of the `add` method, since it would make `logs` inaccessible. Furthermore, `add`'s parameter must point into the same region as the receiver, since `add` makes `log` reachable from `logs`. To express those requirements, we introduce two additional annotations: `@transient` and `@peer`. They are used to annotate the `add` method as follows.

```
class LogList {
    var elem: LogFile = null
    var next: LogList = this
    @transient def add(file: LogFile @peer(this)) =
        if (isEmpty) { elem = file; next = new LogList }
        else next.add(file)
}
```

Note that the `@transient` annotation applies to the receiver, i.e., `this`. `@transient` is equivalent to `@unique`, except that it does not consume the capability to access the annotated parameter (including the receiver). Consequently, it is illegal to pass a transient parameter, or any object reachable from it, to a method expecting a unique parameter, which would consume its capability.

The `@peer(this)` annotation on the parameter type indicates that `file` points into the same region as `this`. The effect on available capabilities is determined by the argument of `@peer`: since `this` is transient, invoking `add` does not consume the capability of `file`.

Note that our system does not restrict references between objects inside the same region; this means that `this` and `file` can refer to each other in arbitrary ways. In the type system this is expressed by having field selections propagate guards: if `this` has

type $\rho \triangleright \text{LogList}$, then `this.elem` has type $\rho \triangleright \text{LogFile}$. Since `file` is a peer of `this`, its type is $\rho \triangleright \text{LogFile}$; therefore, assigning `file` to `elem` in the then-branch of the conditional expression is safe.

To verify the safety of calling `add` in the else-branch, we have to check that `next` and `file` have types guarded by the same capability. Moreover, this capability must be available. Since both conditions are true (the receiver of `isEmpty` is transient), the invocation type-checks.

We have introduced the `@transient` annotation to express the fact that a method maintains the uniqueness and accessibility of a receiver or parameter. The `@peer` annotation indicates that certain parameters are in the same (logical) region of the heap, which allows creating reference paths between them. Together, these annotations enable methods to mutate unique objects without destroying their uniqueness. In the following section we show how the disjoint regions of two unique objects can be merged.

Merging regions. Recall that the parameter of the `add` method shown above is marked as `@peer(this)`, which means that it must be in the same region as the receiver. However, when using `add` in the example of Figure 2, the `log` variable is separately-unique; this means it is contained in a region that is disjoint from the region of `logs`, the receiver of the method call. This is reflected in the types: `log` and `logs` have types $\rho \triangleright \text{LogFile}$ and $\delta \triangleright \text{LogList}$, respectively, for some capabilities $\rho \neq \delta$. Therefore, the invocation `logs.add(log)` is not type-correct. What we need is a way to merge the regions of `log` and `logs` prior to invoking `add`.⁵

In our system, regions are merged using a capture expression of the form `capture(t1, t2)`. The arguments of `capture` must have guarded types $\rho_1 \triangleright T_1$ and $\rho_2 \triangleright T_2$, respectively, such that ρ_1 is available. Our goal is to merge the regions ρ_1 and ρ_2 in a way that (still) permits separately-unique references into region ρ_2 , while giving up the disjointness from region ρ_1 . For this, `capture` returns an alias of t_1 , but with a type guarded by ρ_2 instead of ρ_1 . This allows t_1 and t_2 to refer to each other subsequently. To satisfy the Capability Type Invariant (Definition 3), `capture` consumes ρ_1 . This ensures that t_1 can no longer be accessed under a type guarded by ρ_1 . Therefore, it is safe to break the separation of t_1 and t_2 subsequently. Since ρ_2 is still available, it is possible for separately-unique variables to point into region ρ_2 .

In the example, we use `capture` to merge the regions of `log` and `logs` before invoking `add`:

```
logs.add(capture(log, logs))
```

Note that `capture` consumes the capability of `log`, while the capability of `logs` remains available. The result of `capture` is an alias of `log` in the same region as `logs`. Therefore, the precondition of `add` (see above) is satisfied.

Unique fields. In the example of Figure 2, we made the simplifying assumption that the list of log files is stored in a local variable. This is not the case in the original

⁵ This is similar to changing the owner in systems based on ownership; here, ownership of an object is transferred from one (usually a special “unique”) owner to another [12].

program, where the log files are stored in a field of the class containing the `runTests` method. The main reason is that the lexical scope of `runTests` is too restrictive. It is simpler to create the log file in a method transitively called by `runTests`, at a point where more information about the test is available, and close to the point where the log file is actually used. Consequently, updating the list of log files inside `runTests` would be cumbersome, since it would require returning the log file back into the context of `runTests`. Keeping the `logs` in a field avoids passing it around using (extra) method parameters.

In our system, unique fields must be accessed using an expression of the form `swap($t_1.l, t_2$)`; it returns the current value of the field $t_1.l$ and updates it with the new value t_2 . The first argument must select a unique field. The second argument must be a unique object to be stored as the new value in the field. The object that `swap` returns is always unique, guarded by a fresh capability. The capability of the second argument is consumed, which makes it separately-unique.

In our example, the list of log files can be maintained in a unique field `logFiles` as follows.

```
val logs: LogList @unique = swap(this.logFiles, null)
logs.add(capture(log, logs))
swap(this.logFiles, logs)
```

First, we obtain the current value of the unique `logFiles` field, providing `null` as its new (dummy) value.⁶ Then, we add the `log` file to `logs`, maintaining the uniqueness of `logs` as we discussed above. Finally, we use a second `swap` to update `logFiles` with the modified `logs`.

We now extend the alias invariant introduced above to unique fields. The only way to obtain a reference to an object stored in a unique field is to use the `swap` expression that we just introduced. Therefore, a property that holds for all (references to) objects returned by `swap` is an invariant of unique fields in our system. This allows us to formulate a unique fields invariant that is pleasingly simple.

Definition 4 (Unique Fields Invariant) *References returned by `swap` are separately-unique.*

3 Formalization

This section presents a formalization of our type system. To simplify the presentation of key ideas, we present our type system in the context of a core subset of Java. We add the `capture` and `swap` expressions introduced in the previous section, and augment the type system with capabilities to enforce uniqueness and aliasing constraints. Our approach, however, extends to the whole of Java and other languages like Scala. We discuss important extensions in Section 5.

⁶ Note that it is always safe to treat literals as unique values.

$P ::= \overline{cdef} t$	program
$cdef ::= \text{class } C \text{ extends } D \{ \overline{fld} \overline{meth} \}$	class
$fld ::= \alpha l : C$	field
$meth ::= \text{def } m[\delta](\overline{x} : \overline{T}) : (T, \Delta) = e$	method
$t ::=$	terms
$\text{let } x = e \text{ in } t$	let binding
$y.l := z$	field assignment
y	variable
$e ::=$	expressions
$\text{new } C(\overline{y})$	instance creation
$y.l$	field selection
$y.m(\overline{z})$	method invocation
$\text{capture}(y, z)$	region capture
$\text{swap}(y.l, z)$	unique field swap
t	term
$C, D \in \text{Classes}$	$x, \overline{y}, \overline{z} \in \text{Vars}$
$l \in \text{Fields}$	$\alpha \in \{\text{var, unique}\}$
$m \in \text{Methods}$	$T ::= \rho \triangleright C$
	$\Delta ::= \cdot \mid \Delta \oplus \rho$
	$\rho \in \text{Caps}$

Fig. 4. Core language syntax

Syntax. Figure 4 shows the core language syntax. The syntax of programs, classes, terms, and expressions is standard, except for the `capture` and `swap` expressions, which are new. A program consists of a sequence of class definitions followed by a single top-level term. (We use the common over-bar notation [29] for sequences.) Class definitions consist of declaring a single super-class followed by a body containing field and method definitions. Field definitions carry an additional modifier α , which indicates whether the field points to a unique object ($\alpha = \text{unique}$), or not ($\alpha = \text{var}$). Method definitions are extended with two additional capability annotations that we explain below. The term language is mostly standard. However, note that terms are written in A-normal form [22]: all sub-expressions are variables and the result of each expression is immediately stored into a field or bound in a let. x, y, z are local variables and $x \neq \text{this}$.

Types and capabilities. In our system, there are only guarded types and method types. Guarded types T are composed of an atomic capability ρ and the name of a class. ρ can be seen as the static representation of a region of the heap that contains all objects of a type guarded by ρ . A compound capability Δ is a set of atomic capabilities.

Method types are extended with capabilities δ and Δ . Roughly, Δ indicates which arguments become inaccessible at the call site when the method is invoked; δ is the capability of the result type if it is fresh. The annotations introduced in Section 2 correspond to method types in our core language as follows. A parameter x of type C marked as `@unique` or `@transient` is mapped to a guarded type $\rho \triangleright C$, where ρ is distinct from the capabilities guarding other parameter types. If x is `@transient`, the method returns ρ , i.e., $\rho \in \Delta$. If x is `@unique`, the method consumes ρ , i.e., $\rho \notin \Delta$. A parameter y of type D marked as `@peer(x)` is mapped to a guarded type $\rho' \triangleright D$ if x 's type is guarded by ρ' . `@peer` has no influence on Δ . The receiver (`this`) is treated like a parameter. The

$$\begin{array}{ll}
H ::= \emptyset \mid (H, r \mapsto C(\bar{r})) & \text{heap} \\
V ::= \emptyset \mid (V, y \mapsto \beta \triangleright r) & \text{envir. } (y \notin \text{dom}(V)) \\
R ::= \emptyset \mid R \oplus \beta & \text{dynamic capability}
\end{array}
\quad
\begin{array}{ll}
r \in \text{RefLocs} & \text{reference location} \\
\beta \in \text{DynCaps} & \text{atomic dyn. capability}
\end{array}$$

Fig. 5. Syntax for heaps, environments, and dynamic capabilities

capability δ is distinct from the capabilities of parameters. If the result type is marked `@unique`, its type is guarded by δ . We have $\delta \in \Delta$ only if the result type is guarded by δ , otherwise δ is unused. An unannotated method in the setting of Section 2 has the following type in our core language: the parameters (including `this`) and the result are guarded by the same capability ρ that the method does not consume ($\rho \in \Delta$).

Note that the mapping we just described establishes a precise correspondence: all types expressible in the core language can be expressed using the annotation system of Section 2. This ensures that the formal model is not more powerful than our implemented system.

3.1 Dynamic semantics

We formalize the dynamic semantics in the form of small-step reduction rules. Reduction rules are written in the form $H, V, R, t \longrightarrow H', V', R', t'$. Terms t are reduced in a context consisting of a heap H , a variable environment V , and a set of (dynamic) capabilities R . Figure 5 shows their syntax. A heap maps reference locations to class instances. An instance $C(\bar{r})$ stores location r_i in its i -th field. An environment maps variables to guarded reference locations $\beta \triangleright r$. Note that we do not model explicit stack frames. Instead, method invocations are “flattened” by renaming the method parameters before binding them to their argument values in the environment (as in LJ [41]).

We use the following notational conventions. $R \oplus \beta$ is a short hand for the disjoint union $R \sqcup \{\beta\}$. We define $R \oplus \bar{\beta} := R \oplus \beta_1 \oplus \dots \oplus \beta_n$ where $\bar{\beta} = \beta_1, \dots, \beta_n$.

According to the grammar in Figure 4, expressions are always reduced in the context of a let-binding, except for field assignments. Each operand of an expression is a variable y that the environment maps to a guarded reference location $\beta \triangleright r$. Reducing an expression containing y requires β to be present in the set of capabilities. Since the environment is a flat list of variable bindings, let-bound variables must be alpha-renamable: $\text{let } x = e \text{ in } t \equiv \text{let } x' = e \text{ in } [x'/x]t$ where $x' \notin FV(t)$. (We omit the definition of the FV function to obtain the free variables of a term, since it is completely standard [38].)

The top-level term of a program is reduced in the initial configuration $(r \mapsto \text{Object}(\epsilon)), (\text{this} \mapsto \rho \triangleright r), \{\rho\}$. In the following we explain the reduction rules. The congruence rule for `let` is omitted, since it is standard; `let` $x = y$ in t is reduced in the obvious way.

$$\begin{array}{c}
\frac{V(y) = \delta \triangleright r \quad \delta \in R}{H(r) = C(\bar{r})} \\
\hline
\frac{H, V, R, \text{let } x = y.l_i \text{ in } t}{\longrightarrow H, (V, x \mapsto \delta \triangleright r_i), R, t} \quad (\text{R-SELECT})
\end{array}
\quad
\begin{array}{c}
\frac{V(y) = \delta \triangleright r \quad V(z) = \delta \triangleright r' \quad \delta \in R}{H(r) = C(\bar{r})} \\
\hline
\frac{H' = H[r \mapsto C([r'/r_i]\bar{r})]}{H, V, R, y.l_i := z \longrightarrow H', V, R, y} \quad (\text{R-ASSIGN})
\end{array}$$

The result of selecting a field of a variable y is guarded by the same capability as y . Intuitively, this means that objects transitively reachable from y can only be accessed using variables guarded by the same capability as y . We make this intuition more precise in Section 3.2 where we formalize the separation invariant of Section 2. Assigning to a field requires the variable whose field is updated and the right-hand side to be guarded by the same capability. The heap changes in the standard way.

$$\frac{V(\bar{y}) = \overline{\beta \triangleright r} \quad H' = (H, r \mapsto C(\bar{r})) \quad r \notin \text{dom}(H) \quad \gamma \text{ fresh}}{H, V, R \oplus \overline{\beta}, \text{let } x = \text{new } C(\bar{y}) \text{ in } t \longrightarrow H', (V, x \mapsto \gamma \triangleright r), R \oplus \gamma, t} \text{ (R-NEW)}$$

Creating a new instance consumes the capabilities of the constructor arguments. This ensures that the arguments are effectively separately-unique. Consequently, it is safe to assign (some of) the arguments to unique fields of the new instance. In our core language, creating a new instance always yields a unique object. Therefore, the new let-bound variable that refers to it is guarded by a fresh capability.

$$\frac{\begin{array}{c} V(y) = \beta_1 \triangleright r_1 \quad H(r_1) = C_1(_) \\ V(\bar{z}) = \beta_2 \triangleright r_2 \dots \beta_n \triangleright r_n \quad \overline{\beta} \subseteq R \\ mbody(m, C_1) = (\bar{x}, e) \end{array}}{\begin{array}{c} H, V, R, \text{let } x = y.m(\bar{z}) \text{ in } t \\ \longrightarrow H, (V, x \mapsto \beta \triangleright r), R, \text{let } x = e \text{ in } t \end{array}} \text{ (R-Invoke)}$$

The rule for method invocation uses a standard auxiliary function $mbody$ to obtain the body of a method. It is defined as follows. Let $\text{def } m[\delta](x : T) : (R, \Delta) = e$ be a method defined in the most direct super-class of C that defines m . Then $mbody(m, C) = (\bar{x}, e)$.

$$\frac{\begin{array}{c} V(y) = \beta \triangleright r \quad V(z) = \gamma \triangleright _ \\ \overline{\beta} \subseteq \gamma \end{array}}{\begin{array}{c} H, V, R \oplus \beta \oplus \gamma, \text{let } x = \text{capture}(y, z) \text{ in } t \\ \longrightarrow H, (V, x \mapsto \gamma \triangleright r), R \oplus \gamma, t \end{array}} \text{ (R-CAPTURE)}$$

Capture merges the regions of its two arguments y and z . It returns an alias of y guarded by the capability of z . This allows storing a reference to y in a field of z and vice versa (see rule (R-ASSIGN) above). By consuming y 's capability, we make sure that objects that used to be in region β remain accessible only through variables guarded by γ , which is the capability of z . This enforces that all objects are accessible as part of at most one region at a time. (Recall that variables whose capabilities are not available cannot be accessed.)

$$\frac{\begin{array}{c} V(y) = \beta \triangleright r \quad H(r) = C(\bar{r}) \quad \gamma \text{ fresh} \\ V(z) = \beta' \triangleright r' \quad H' = H[r \mapsto C([r'/r_i]\bar{r})] \end{array}}{\begin{array}{c} H, V, R \oplus \beta \oplus \beta', \text{let } x = \text{swap}(y, l_i, z) \text{ in } t \\ \longrightarrow H', (V, x \mapsto \gamma \triangleright r_i), R \oplus \beta \oplus \gamma, t \end{array}} \text{ (R-SWAP)}$$

The only way to access a unique field is using swap . It mutates a unique field to point to a new object, and returns the field's previous value. The first argument must select a

unique field such that the capability of the containing object is available. The second argument must be guarded by a different capability, which is consumed. This ensures that the new value and the object containing the unique field are separate prior to evaluating `swap`. `swap` returns the field's old value; the new let-bound variable that refers to it is guarded by a fresh capability, which allows treating the variable as separately-unique.

3.2 Static semantics

Well-formed programs. A program is well-formed if all its class definitions are well-formed. Classes and methods are well-formed according to the following rules. (We write \dots to omit unimportant parts of code in a program P .)

$$\frac{\begin{array}{c} C \vdash \overline{meth} \\ D = \text{Object} \vee P(D) = \text{class } D \dots \\ \forall (\text{def } m \dots) \in \overline{meth}. \text{override}(m, C, D) \\ \forall \alpha l : E \in \overline{fld}. l \notin \text{fields}(D) \end{array}}{\vdash \text{class } C \text{ extends } D \{ \overline{fld} \overline{meth} \}} \quad (\text{WF-CLASS})$$

All well-formed class hierarchies are rooted in `Object`. All methods in a well-formed class definition are well-formed. We explain well-formed method overriding below. Fields may not be overridden; their names must be different from the names of fields in super-classes. We use a standard function $\text{fields}(D)$ [29] to obtain all fields in D and super-classes of D .

$$\frac{\begin{array}{c} \overline{T} = \rho \triangleright C \quad \overline{x : T} ; \{\rho \mid \rho \in \overline{\rho}\} \vdash e : R ; \Delta \\ x_1 = \text{this} \quad R = \delta' \triangleright D \quad \delta' \in \Delta \\ \delta = \begin{cases} \delta' & \text{if } \delta' \notin \overline{\rho} \\ \text{fresh otherwise} & \end{cases} \end{array}}{C_1 \vdash \text{def } m[\delta](\overline{x : T}) : (R, \Delta) = e} \quad (\text{WF-METHOD})$$

In a well-formed method definition that appears in class C_1 , the first parameter is always `this` and its class type is C_1 . The method body must be type-checkable in an environment that binds the parameters to their declared types, and that provides all capabilities of the parameter types. After type-checking the body, the capabilities in Δ must still be available. The result type of a method must be guarded by a capability in Δ . If the capability of the result type does not guard one of the parameter types, it is unknown in the caller's context. In this case we treat it as existentially quantified; the square brackets are used as its binder. If the capability of the result type guards one of the parameter types, the quantified capability is unused.

$$\frac{\begin{array}{c} \text{mtype}(m, D) \text{ not defined} \vee \\ (\text{mtype}(m, D) = \exists \delta. (\rho \triangleright D, \overline{T}) \rightarrow (R, \Delta) \wedge \\ \text{mtype}(m, C) = \exists \delta. (\rho \triangleright C, \overline{T}) \rightarrow (R, \Delta)) \end{array}}{\text{override}(m, C, D)} \quad (\text{WF-OVERRIDE})$$

A method defined in class C satisfies the rule for well-formed overriding if the super-class D does not define a method of the same name, or the method types differ only in the first `this` parameter.

Subclassing and subtypes. Each program defines a class table, which defines the sub-typing relation $<:$. In our system, $<:$ is identical to that of FJ [29], except for the following rule for guarded types, which is new. It expresses the fact that guarded types can only be sub-types if their capabilities are equal.

$$\frac{C <: D}{\rho \triangleright C <: \rho \triangleright D} \quad (\text{<:-CAP})$$

Type assignment. Terms are type-checked using the judgement $\Gamma ; \Delta \vdash t : T ; \Delta'$. Γ maps variables to their types. The facts that Γ implies can be used arbitrarily often in typing derivations. Δ and Δ' are capabilities, which may not be duplicated. As part of the typing derivation, capabilities may be consumed or generated. Δ' denotes the capabilities that are available after deriving the type of the term t . In a typing derivation where $\Delta' = \Delta$ we omit Δ' for brevity.

$$\frac{\Gamma(y) = \rho \triangleright C \quad \rho \in \Delta}{\Gamma ; \Delta \vdash y : \rho \triangleright C ; \Delta} \quad (\text{T-VAR}) \quad \frac{\Gamma ; \Delta \vdash y : \rho \triangleright C \quad \text{fields}(C) = \overline{\alpha l : D} \quad \alpha_i \neq \text{unique}}{\Gamma ; \Delta \vdash y.l_i : \rho \triangleright D_i ; \Delta} \quad (\text{T-SELECT})$$

A variable is well-typed in Γ, Δ if Γ contains a binding for it, and Δ contains the capability of its (guarded) type. This ensures that the capabilities of variables occurring in a typing derivation are statically available. Selecting a field from a variable y of guarded type yields a type guarded by the same capability. The selected field must not be unique. Because of rule (T-VAR), the capability of y must be available.

$$\frac{\Gamma ; \Delta \vdash y : \rho \triangleright C \quad \Gamma ; \Delta \vdash z : \rho \triangleright D_i \quad \text{fields}(C) = \overline{\alpha l : D} \quad \alpha_i \neq \text{unique}}{\Gamma ; \Delta \vdash y.l_i := z : \rho \triangleright C ; \Delta} \quad (\text{T-ASSIGN})$$

Assigning to a non-unique field of a variable y with guarded type $\rho \triangleright C$ requires also the right-hand side to be guarded by ρ . The term has the same type as y , which is the result of reducing the assignment (see Section 3.1).

$$\frac{\Gamma ; \Delta \vdash \overline{y : \rho \triangleright D} \quad \Delta = \Delta' \oplus \bar{\rho} \quad \text{fields}(C) = \overline{\alpha l : D} \quad \rho' \text{ fresh}}{\Gamma ; \Delta \vdash \text{new } C(\bar{y}) : \rho' \triangleright C ; \Delta' \oplus \rho'} \quad (\text{T-NEW})$$

The rule for instance creation requires all constructor arguments to be guarded by distinct capabilities, which must be available. Intuitively, this means that the arguments are in mutually disjoint regions. Therefore, it is safe to assign them to unique fields of

the new instance. By consuming the capabilities of the arguments, we ensure that there is no usable reference left that could point into the object graph rooted at the new instance; thus, we can assign a type guarded by a fresh capability ρ' to the new instance and make ρ' available to the context. Note that we can relax this rule for initializing non-unique fields: multiple non-unique fields may be guarded by the same capability. (See Section 5 for a discussion in the context of nested classes.)

$$\frac{\begin{array}{c} \Gamma ; \Delta \vdash y : \rho_1 \triangleright D_1 \\ \Gamma ; \Delta \vdash z_{i-1} : \rho_i \triangleright D_i, i = 2..n \\ mtype(m, D_1) = \exists \delta. \delta \triangleright D \rightarrow (R, \Delta_m) \\ \sigma = \overline{\delta \mapsto \rho \circ \delta \mapsto \rho} \text{ injective} \quad \rho \text{ fresh} \\ \Delta = \Delta' \uplus \{\rho \mid \rho \in \overline{\rho}\} \end{array}}{\Gamma ; \Delta \vdash y.m(\overline{z}) : \sigma R ; \sigma \Delta_m \oplus \Delta'} \quad (\text{T-Invoke})$$

In the rule for method invocations, the capabilities of all arguments must be available in Δ . We look up the method type based on the static type of the receiver. The capabilities in method types are abstract, and have to be instantiated with concrete ones. To satisfy the pre-condition of the method, there must be a substitution that maps the capabilities of the formal parameters to the capabilities of the arguments. Importantly, the substitution must be *injective* to prevent mapping different formal capabilities to the same argument capability; this would mean that the requirement to have two different formal capabilities could be met using only a single argument capability, which would amount to duplicating that capability. In our system, capabilities may never be duplicated. The resulting set of capabilities is composed of the capabilities provided by the method after applying the substitution ($\sigma \Delta_m$) and those capabilities Δ' that were provided by the context, but that were not required by the method.

$$\frac{\Gamma ; \Delta \vdash y : \rho \triangleright C \quad \Gamma ; \Delta \vdash z : \rho' \triangleright C' \quad \Delta = \Delta' \oplus \rho}{\Gamma ; \Delta \vdash \text{capture}(y, z) : \rho' \triangleright C ; \Delta'} \quad (\text{T-CAPTURE})$$

The type rule for `capture` requires the capabilities of the arguments to be present (this follows from (T-VAR), see above). The capability of the first argument is consumed, thereby making all variables pointing into its region inaccessible. The result has the same class type as y , but guarded by the capability of z . Essentially, `capture` casts its first argument from its current region to the region of the second argument; in Section 4 we prove that the cast can never fail at run-time.

$$\frac{\begin{array}{c} \Gamma ; \Delta \vdash y : \rho \triangleright C \quad \Gamma ; \Delta \vdash z : \rho' \triangleright D_i \\ \text{fields}(C) = \overline{\alpha_i l : D} \quad \alpha_i = \text{unique} \\ \Delta = \Delta' \oplus \rho' \quad \rho'' \text{ fresh} \end{array}}{\Gamma ; \Delta \vdash \text{swap}(y.l_i, z) : \rho'' \triangleright D_i ; \Delta' \oplus \rho''} \quad (\text{T-SWAP})$$

The first argument of `swap` must select a unique field. Recalling the dynamic semantics, `swap` returns the current value of this field, and assigns the value of z to it. Therefore, the field must have the same class type as z (possibly using subsumption, see below).

The arguments must be guarded by two different capabilities, which must be present in Δ . (Again, ρ is present because of (T-Var).) This means that the arguments point to disjoint regions in the heap. By consuming the capability of z , we ensure that it is separately-unique. Since the reference returned by `swap` is unique, the result is guarded by a fresh capability.

$$\frac{\Gamma ; \Delta \vdash e : T ; \Delta' \quad \Gamma, x : T ; \Delta' \vdash t : T' ; \Delta''}{\Gamma ; \Delta \vdash \text{let } x = e \text{ in } t : T' ; \Delta''} \quad (\text{T-LET}) \qquad \frac{\Gamma ; \Delta \vdash e : T' ; \Delta' \quad T' <: T}{\Gamma ; \Delta \vdash e : T ; \Delta'} \quad (\text{T-SUB})$$

The rule for let is standard, except for the fact that type derivations may change the set of capabilities. The subsumption rule can be applied wherever the type of an expression is derived. In particular, deriving the type of variables is subject to subsumption.

Well-formedness. We require terms to be reduced in well-formed configurations. A well-formed configuration must satisfy at least the following two invariants, which are central to the soundness of our system. The first invariant expresses the fact that two accessible variables guarded by different capabilities do not share a common reachable object.

Definition 1 (Separation Invariant) A configuration V, H, R satisfies the Separation Invariant, written $\text{separation}(V, H, R)$, iff

$$\begin{aligned} & \forall (x \mapsto \delta \triangleright r), (x' \mapsto \delta' \triangleright r') \in V. \\ & (\delta \neq \delta' \wedge \{\delta, \delta'\} \subseteq R \Rightarrow \text{separate}(H, r, r')) \end{aligned}$$

Note that we can only conclude that the two variables are separate if both capabilities are present. In particular, capturing a variable y does not violate the invariant even though it creates an alias of y guarded by a different capability. The reason is that `capture` consumes y 's capability, thereby making it inaccessible. Therefore, the invariant continues to hold for accessible variables, that is, variables whose capabilities are present. (The predicate `separate` is formally defined in the companion technical report [24].)

Definition 2 (Unique Fields Invariant) A configuration V, H, R satisfies the Unique Fields Invariant, written $\text{uniqFlds}(V, H, R)$, iff

$$\begin{aligned} & \forall (x \mapsto \delta \triangleright r) \in V. H(q) = C(\bar{p}) \Rightarrow \forall i \in \text{uniqInd}(C). \\ & \delta \in R \wedge \text{reachable}(H, p_i, r') \Rightarrow \text{domedge}(H, q, i, r, r') \end{aligned}$$

The unique fields invariant says that all reference paths from a variable x to some object r' reachable from a unique field must “go through” that unique field. The `reachable` and `domedge` predicates are based on the following definition of reference paths.

$$\frac{r \in \text{dom}(H)}{[r] \in \text{path}(H, r, r)} \qquad \frac{H(r) = C(\bar{p}) \quad \exists i. P \in \text{path}(H, p_i, r')}{r :: P \in \text{path}(H, r, r')} \qquad \frac{\text{path}(H, r, r') \neq \emptyset}{\text{reachable}(H, r, r')}$$

Basically, a reference path is a sequence of reference locations, where each reference (except the first) is stored in a field of the preceding location. The definition of *domedge* is as follows.

$$\begin{aligned} \text{domedge}(H, q, i, r, r') &\Leftrightarrow \\ \forall P \in \text{path}(H, r, r'). P = r \dots q, p_i, \dots r' \text{ where } H(q) = C(\bar{p}) \end{aligned}$$

This predicate expresses the fact that all paths from r to r' must contain the sequence q, p_i , which corresponds to selecting the i -th (unique) field of object q .

$$\frac{\begin{array}{c} \Sigma \vdash H \\ \Gamma ; \Delta ; \Sigma \vdash V ; R \\ \text{separation}(V, H, R) \\ \text{uniqFlds}(V, H, R) \end{array}}{\Gamma ; \Delta ; \Sigma \vdash H ; V ; R} \quad (\text{WF-CONFIG})$$

Aside from the separation and unique fields invariants, well-formed configurations must have well-formed environments and heaps.

$$\frac{\begin{array}{c} \Sigma \vdash H \quad \Sigma(r) = C \\ \text{fields}(C) = \alpha l : D \quad \Sigma \vdash p : D \end{array}}{\Sigma \vdash (H, r \mapsto C(\bar{p}))} \quad (\text{WF-HEAP}) \quad \frac{\Sigma(r) = D \quad D <: C}{\Sigma \vdash r : C} \quad (\text{HEAP-TYPE})$$

The rule for well-formed heaps is completely standard: the heap typing Σ must agree with the heap H on the type of each class instance. Moreover, the types of instances referred to from its fields must be compatible with their declared types using the (HEAP-TYPE) rule.

$$\frac{\begin{array}{c} \Gamma ; \Delta ; \Sigma \vdash V ; R \\ \Sigma \vdash r : C \quad \rho \in \Delta \text{ iff } \beta \in R \end{array}}{(\Gamma, y : \rho \triangleright C) ; \Delta ; \Sigma \vdash (V, y \mapsto \beta \triangleright r) ; R} \quad (\text{WF-ENV})$$

In the rule for well-formed environments we require the type environment Γ to agree with the heap typing Σ on the class type of instances referred to from variables. This rule also contains the key to relating static and dynamic capabilities: the static capability of a variable is contained in the set of static capabilities if and only if its dynamic capability in the environment is contained in the set of dynamic capabilities. This precise correspondence allows us to prove that the reduction of a well-typed term will never get stuck because of missing capabilities (see Section 4).

4 Soundness

In this section we present the main soundness result for the type system introduced in Section 3. We prove type soundness using the standard syntactic approach of preservation plus progress [47]. A complete proof of soundness appears in the companion technical report [24].

In a first step, we prove a preservation theorem: it states that the reduction of a well-typed term in a well-formed context preserves the term's type. Moreover, the resulting context (heap, environment, and capabilities) is well-formed with respect to a new type environment, static capabilities, and heap typing.

Theorem 1 (Preservation) *If*

- $\Gamma ; \Delta \vdash t : T ; \Delta'$
- $\Gamma ; \Delta ; \Sigma \vdash H ; V ; R$
- $H, V, R, t \longrightarrow H', V', R', t'$

then there are $\Gamma' \supseteq \Gamma$, $\Delta'' \supseteq \Delta'$, and $\Sigma' \supseteq \Sigma$ such that

- $\Gamma' ; \Delta'' \vdash t' : T ; \Delta'$
- $\Gamma' ; \Delta'' ; \Sigma' \vdash H' ; V' ; R'$

This theorem guarantees that reduction preserves the separation and unique fields invariants that we introduced in Section 3.2. These invariants are implied by the well-formedness of the result context H', V', R' . Also note that the new type environment Γ' and the new heap typing Σ' are super sets of their counterparts Γ and Σ , respectively. This means that merging two regions does not require strong type updates in our system.

In a second step, we prove a progress theorem, which guarantees that a well-typed term can be reduced in a well-formed context, unless it is a value. Variables are the only values in our language.

Theorem 2 (Progress) *If $\Gamma ; \Delta \vdash t : T ; \Delta'$ and $\Gamma ; \Delta ; \Sigma \vdash H ; V ; R$, then either $t = y$, or there is a reduction $H, V, R, t \longrightarrow H', V', R', t'$.*

The progress theorem makes sure that the reduction of a term does not get stuck, because of missing capabilities; that is, if a term type-checks, all required capabilities will be available during its reduction. Soundness of the type system follows from Theorem 1 and Theorem 2.

We can formulate a uniqueness theorem as a corollary of preservation and progress. Formally, separate uniqueness is defined as follows:

Definition 3 (Separate Uniqueness) *A variable $(y \mapsto \beta \triangleright r) \in V$ such that $\beta \in R$ is separately-unique in configuration H, V, R , let $x = t$ in t' iff*

$$\forall (y' \mapsto \beta' \triangleright r') \in V. (\neg\text{separate}(H, r, r') \wedge H, V, R, t \longrightarrow^* H', V', R', e') \Rightarrow \beta' \notin R'$$

Intuitively, the definition says that the capabilities of aliases of a variable y are unavailable when reducing t' if y is separately-unique in t . By Theorem 2 and the reduction rules, none of y 's aliases are accessed after the reduction of t , since $\beta' \notin R'$.

The following corollary guarantees that a variable passed as an argument to a method expecting a unique parameter is separately-unique; this means that all variables that are still accessible after the invocation are separate from the argument.

Corollary 1 (Uniqueness) *If*

- $\Gamma ; \Delta \vdash \text{let } x = t \text{ in } t' : T ; \Delta' \text{ where } t = y.m(\bar{z})$
- $\Gamma ; \Delta ; \Sigma \vdash H ; V ; R$
- $\Gamma(y) = _ \triangleright C$
- $mtype(m, C) = \exists \delta. \overline{\delta \triangleright D} \rightarrow (T_R, \Delta_m) \text{ where } \delta_i \notin \Delta_m$

then z_i is separately-unique in $H, V, R, \text{let } x = t \text{ in } t'$.

5 Extensions

In this section we address some of the issues when integrating our type system into full-featured languages like Scala or Java that we omitted from the formalization for simplicity.

Closures. A number of object-oriented languages, such as Scala, have special support for closures. In this section we discuss how our type system handles closures that capture unique variables in their environment.

Consider the following example: a unique list of books should be inserted into a hash map in a way that enables fast access to the books in a certain category. The following code achieves this in an efficient way.

```
val list: List[Book] @unique = ...
val map = new HashMap[String, List[Book]]
list.foreach { b =>
  val sameCat = map(b.cat)
  map.put(b.cat, b :: sameCat)
}
```

For safety, we require the produced hash map to be unique. This means that the capability of `map` must be available after the `foreach`. Intuitively, this is the case if `list` and `map` are in the same region; the body of `foreach` only adds references from the hash map to the books.

Technically, the closure is type-checked as follows. First, we collect the capabilities of references that the closure captures. We require all captured references to be guarded by the same capability, say, ρ . The reason is that all of these references are stored in the same (closure) object, which must, therefore, be guarded by ρ .⁷ In a second step, the body is type-checked assuming that the closure's parameters are also guarded by ρ . In addition, we require that ρ is not consumed in the body. This check allows us to associate a single capability, ρ , with the closure. It indicates that ρ must be available when invoking the closure; moreover, arguments must be guarded by ρ . The type of a closure guarded by ρ , written $\rho \triangleright (A \Rightarrow B)$, effectively corresponds to the method type $\rho \triangleright A \rightarrow (\rho \triangleright B, \{\rho\})$.

Revisiting our example, the `foreach` method in class `List` can then be annotated and type-checked as follows.

⁷ Captured references guarded by different capabilities would have to be stored in unique fields of the closure object; accessing them would require `swap`. Currently, we do not see a practical way to support that.

```
@transient def foreach(f: (A => Unit) @peer(this)) {
  if (!this.isEmpty) { f(this.head); this.tail.foreach(f) }
}
```

Here, `f`'s argument, `this.head`, must be guarded by the same capability as `f`; this is the case, since `f` is a peer of `this`. It is important to note that this does not break any existing code: the annotations merely express that the receiver and the variables captured by `f` must be in the same region. Unannotated objects of existing clients are (all) contained in the global “shared” region, and are therefore compatible with the annotated `foreach`.

What if a closure does not capture a variable in the environment? In this case, we assume that the closure’s parameters are guarded by some fresh capability, say, δ , when checking the body. When a closure of type $\delta \triangleright (A \Rightarrow B)$ is passed to a method invoked on an object guarded by ρ , we first capture the closure; this yields a reference to the closure with type $\rho \triangleright (A \Rightarrow B)$, consuming δ . Note that this capturing can occur implicitly, without additions to the program.

Nested classes. Nested classes can be seen as a generalization of closures; a nested class may define multiple methods, and it may be instantiated several times. An important use case are anonymous iterator definitions in collection classes.

For instance, the `SingleLinkedList` class in Scala’s standard library provides the following method for obtaining an iterator (the `A` type parameter is the collection’s element type):

```
def elements: Iterator[A] = new Iterator[A] {
  var elems = SingleLinkedList.this
  def hasNext = (elems ne null)
  def next = { val res = elems.elem; elems = elems.next; res }
}
```

The nested `Iterator` subclass stores a captured reference to the receiver in its `elems` field. Therefore, the iterator instance cannot be unique, since it is not separate from the receiver. However, it is safe to create the iterator in the region of the receiver.

In general, new nested class instances can be created in the region of captured references if all those references are in the same region, say, ρ (similar to closures). If there are constructor arguments, they must also be guarded by ρ . Note that creating the instance does not consume ρ . This means, we are relaxing the rule for instance creation introduced in Section 3, which requires the capabilities of constructor arguments to be distinct and consumed; it applies equally to non-nested classes. Note that nested classes may have unique fields; initializing unique fields through constructor parameters must follow the same rule as normal instance creation, that is, the arguments must be guarded by distinct capabilities, which are consumed.

Revisiting the iterator example, we can use the `@peer` annotation to express the fact that the iterator is created in the same region as the receiver:

```
@transient def elements: Iterator[A] @peer(this) = ...
```

This enables arbitrary uses of an iterator while the capability of its underlying unique collection is available.

Transient Classes. We say that a class is *transient* if none of its fields are unique and all of its methods can be annotated such that the receiver is marked as `@transient` and all parameters are marked as `@peer(this)`. This means that the receiver and the parameters of a method must be guarded by the same capability. It ensures that neither the receiver nor objects reachable from it are leaked to (potentially) shared objects, since (1) shared objects are guarded by the special “shared” capability, and (2) capabilities of method parameters are universally quantified, making them incompatible with the shared capability. We have found that most classes used in messages are transient (see Section 6). This means that most objects only interact with objects from its enclosing aggregate, which is consistent with the results for thread-locality in Loci [50]. To abbreviate the canonical annotation, we allow classes to be annotated as `@transient`.

6 Implementation

We have implemented our type system as a plug-in for the Scala compiler developed at EPFL.⁸ The plug-in inserts an additional compiler phase that runs right after the normal type checker. The extended compiler first does standard Scala type checking on the erased terms and types of our system. Then, types and capabilities are checked using (an extension of) the type rules presented in Section 3. For subsequent code generation, all capabilities, `capture` and `swap` expressions are erased.

Practical Experience. As a first step we annotated the (mutable) `DoubleLinkedList`, `ListBuffer`, and `HashMap` classes from the collections of Scala 2.7 including all classes/traits that these classes transitively extend, comprising 2046 lines of code. Making all classes transient (see Section 5) required changing 60 source lines.

In Section 2, we have already introduced the partest testing framework, which is used to run the check-in and nightly tests for the Scala compiler and standard library. Although the majority of code deals with compiler/test set-up and reporting, the unique objects that are transferred among actors are used pervasively throughout large parts of the code. An example for such a class is `LogFile`, which receives output from various sources (compiler, test runner etc.). For creating unique `LogFile` instances it is sufficient that the class is transient; However, `LogFile` inherits from the standard `java.io.File` class, which is unchecked. Fortunately, according to the Java version 6 API, “instances of the `File` class are immutable.”⁹ We configured our type checker to skip checking immutable classes. In general, however, this is unsound if such classes could mutate or leak method parameters. Overall, the most important changes were:

1. Annotating the message classes. We found that all message classes could be annotated as `@transient`.
2. Handling of unique fields. We had to annotate a field holding a list of created log files as `@unique`. Three `swap` expressions were sufficient to cover all accesses to the field.

⁸ See <http://lamp.epfl.ch/~phaller/capabilities.html>.

⁹ See <http://java.sun.com/javase/6/docs/api/>.

In summary, out of the 4182 lines of code (including whitespace), we had to change 32 lines and add 29 additional lines. The following observation helped interoperability: passing a unique object to an unannotated method is often unproblematic if the method expects an immutable type. However, this is unsound in the general case, since instances of such types could be downcast to mutable types. In our study we allowed passing a `LogFile` instance to methods of unannotated Java classes expecting a `java.io.File`.

7 Other Related Work

In functional languages, linear types [43] have been used to implement operations like array updating without the cost of a full copy. An object of linear type must be used exactly once; as a result, linear objects must be threaded through the computation. Wadler’s `let!` or observers [34] can be used to temporarily access a linear object under a non-linear type. Linear types have also been combined with regions, where `let!` is only applicable to regions [46]. Bierhoff and Aldrich [6] build on an expressive linear program logic for modular type-state checking in an object-oriented setting. It is not clear how their system could be applied to message-based concurrency, since unique references do not prevent read-only references to the same object. Beckman et al. [5] use a similar system for verifying the correct use of software transactions. JAVA(X) [15] tracks linear and affine resources using type refinement and capabilities, which are structured, unlike ours. The authors did not consider applications to concurrency. Shoal [2] combines static and dynamic checks to enforce sharing properties in concurrent C programs; in contrast, our approach is purely static. Like in region-based memory management [42, 44, 27, 52], in our system objects inside a region may not refer to objects inside another region that may be separately consumed. The main differences are: first, regions in our system do not have to be consumed/deleted, since they are garbage-collected; second, regions in our system can be merged. Separation logic [36] is a program logic designed to reason about separation of portions of the heap; the logic is not decidable and does not deal with stack variables, unlike our approach. Bor nat et al. [7] study permission accounting in separation logic; unlike our system, their approach is not automated. Parkinson and Bierman [37] extend the logic to an object-oriented setting; however, applications [18] still require heavy-weight theorem proving and involve extensive program annotation. To avoid aliasing, swapping [26] has been proposed previously as an alternative to copying pointers; in contrast to earlier work, our approach integrates swapping with internally-aliased unique references and local aliasing.

8 Conclusion

We have introduced a new type-based approach to uniqueness in object-oriented programming languages. Simple capabilities enforce both aliasing constraints for uniqueness and at-most-once consumption of unique references. By identifying unique and borrowed references as much as possible our approach provides a number of benefits: first, a simple formal model, where unique references “subsume” borrowed references.

Second, the type system does not require complex features, such as existential ownership or explicit region declarations. The type system has been proven sound and can be integrated into full-featured languages, such as Scala. Practical experience with collection classes and actor-based concurrent programs suggests that the system allows type checking real-world Scala code with only few changes.

Acknowledgments: Thanks to Rémi Bonnet for discussions on earlier versions of the type system. Thanks to the anonymous reviewers for detailed and helpful comments.

References

1. Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, pages 32–59, 1997.
2. Zachary R. Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *PLDI*, pages 98–109. ACM, 2009.
3. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
4. Joshua S. Auerbach, David F. Bacon, Rachid Guerraoui, Jesper Honig Spring, and Jan Vitek. Flexible task graphs: a unified restricted thread programming model for java. In *LCTES*, pages 1–11. ACM, 2008.
5. Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In Gail E. Harris, editor, *OOPSLA*, pages 227–244. ACM, 2008.
6. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320. ACM, 2007.
7. Richard Bornat, Cristiano Calcagno, Peter W. O’Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.
8. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
9. John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL*, pages 283–295. ACM, 2005.
10. Denis Caromel. Towards a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102, 1993.
11. Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ICFP*, pages 213–224. ACM, 2008.
12. Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200. Springer, 2003.
13. Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *APLAS*, pages 139–154. Springer, 2008.
14. David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
15. Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with java(X). In *ECOOP*, pages 550–574. Springer, 2007.
16. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP*, pages 28–53. Springer, 2007.
17. Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
18. Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for java. In *OOPSLA*, pages 213–226. ACM, 2008.

19. Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *ESOP*, pages 204–218. Springer, 2004.
20. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys*, pages 177–190. ACM, 2006.
21. Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002.
22. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.
23. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
24. Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. Technical Report LAMP-REPORT-2009-004, <http://infoscience.epfl.ch/record/142817>, EPFL, Lausanne, Switzerland, December 2009.
25. Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
26. Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Software Eng.*, 17(5):424–435, May 1991.
27. Michael W. Hicks, J. Gregory Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *ISMM*, pages 73–84, 2004.
28. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, 1991.
29. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
30. Naoki Kobayashi. Quasi-linear types. In *POPL*, pages 29–42, 1999.
31. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
32. Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *OOPSLA*, 2007.
33. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, pages 158–185. Springer, 1998.
34. Martin Odersky. Observers for linear types. In *ESOP*, pages 390–407. Springer, 1992.
35. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, Mountain View, CA, 2008.
36. Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19. Springer, 2001.
37. Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM, 2008.
38. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
39. Jesper Honig Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in java. In *OOPSLA*, pages 211–228, 2007.
40. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP*, pages 104–128. Springer, 2008.
41. Rok Strniša, Peter Sewell, and Matthew J. Parkinson. The java module system: core design and semantic definition. In *OOPSLA*, pages 499–514, 2007.
42. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.
43. Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, Israel, 1990. IFIP TC 2 Working Conference.
44. David Walker, Karl Crary, and J. Gregory Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, 2000.
45. David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *TIC*, pages 177–206. Springer, 2000.
46. David Walker and Kevin Watkins. On regions and linear types. In *ICFP*, 2001.

47. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.
48. Tobias Wrigstad. *Ownership-Based Alias Management*. PhD thesis, KTH, Sweden, 2006.
49. Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4), 2007.
50. Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for java. In *ECOOP*, pages 445–469. Springer, 2009.
51. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA*, pages 258–268, 1986.
52. Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *RTSS*, pages 241–251. IEEE Computer Society, 2004.

A Verified Compiler for an Impure Functional Language

Adam Chlipala

Harvard University, Cambridge, MA, USA

adamc@cs.harvard.edu

Abstract

We present a verified compiler to an idealized assembly language from a small, untyped functional language with mutable references and exceptions. The compiler is programmed in the Coq proof assistant and has a proof of total correctness with respect to big-step operational semantics for the source and target languages. Compilation is staged and includes standard phases like translation to continuation-passing style and closure conversion, as well as a common subexpression elimination optimization. In this work, our focus has been on discovering and using techniques that make our proofs easy to engineer and maintain. While most programming language work with proof assistants uses very manual proof styles, all of our proofs are implemented as adaptive programs in Coq’s tactic language, making it possible to reuse proofs unchanged as new language features are added.

In this paper, we focus especially on phases of compilation that rearrange the structure of syntax with nested variable binders. That aspect has been a key challenge area in past compiler verification projects, with much more effort expended in the statement and proof of binder-related lemmas than is found in standard pencil-and-paper proofs. We show how to exploit the representation technique of *parametric higher-order abstract syntax* to avoid the need to prove any of the usual lemmas about binder manipulation, often leading to proofs that are actually shorter than their pencil-and-paper analogues. Our strategy is based on a new approach to encoding operational semantics which delegates all concerns about substitution to the meta language, without using features incompatible with general-purpose type theories like Coq’s logic.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods; D.3.4 [*Programming Languages*]: Compilers

General Terms Languages, Verification

Keywords compiler verification, interactive proof assistants

1. Introduction

Mechanized proof about programming languages is rather new as an engineering discipline. Only a handful of “real world” projects have been undertaken with users beyond computer science and mathematics researchers. Still, projected practical applications underlie most recent work. For example, compiler verification holds

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL’10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01... \$10.00

the promise of dramatically reducing the costs of quality assurance in the development, evolution, and maintenance of compilers. Unfortunately, this sort of verification seems today to require epic investments of time and cleverness by experts in semantics and theorem-proving. The main message of this paper is that, as in more familiar software development, compiler verification admits design patterns that cut down dramatically on the required grunt work, to the point where it seems plausible that the use of verification can actually reduce the overall effort required to build a correct compiler, even when the baseline we compare against is almost-correct compilers where copious testing has found all but the most obscure bugs.

There has been much important research on verifying compilers for relatively low-level languages like C, including in the verified language stack project by Moore (1989) and the more recent CompCert project (Leroy 2006). In that domain, languages researchers generally start projects and discover that, when they pick the wrong abstractions and proof structuring principles, verification requires much more work than they expected. In contrast, when picking the wrong abstractions in mechanized proofs about languages with nested variable binders (such as most functional languages), the same researchers often find themselves buried in details that they thought of as irrelevant. We have heard many stories of knowledgeable semanticists outright giving up on these kinds of proofs.

Recently, there has been much progress in research on the representation techniques that minimize the chances of such defeats. The use of higher-order abstract syntax (HOAS) in Twelf (Pfenning and Schürmann 1999) remains a popular choice, though it seems that a majority of languages researchers prefer interactive proof assistants that, unlike Twelf, can automate large parts of proofs. The de Bruijn index representation (de Bruijn 1972) is another old standard that sees wide use today. Perhaps the most popular methodologies now center around the nominal logic package for Isabelle/HOL (Urban and Tasson 2005) and the Penn approach to locally nameless binding in Coq (Aydemir et al. 2008). With these techniques, many facts about variable freshness and term well-formedness remain present explicitly in proofs, but there are standard recipes for figuring out the right lemmas to prove and when to apply them.

These recipes make it likely that a user with enough perseverance will manage to prove his theorem. This is a great improvement over the situation of just a few years ago, but, in this paper, we argue that we should be striving for more. In software engineering, we focus on maximizing programmer productivity, and we believe that mechanized proof engineering could stand to see a similar focus.

Most parts of most proofs about practical programming languages are exercises in stepping through many cases that are proved in unenlightening ways, with a handful of cases representing the core insights of a proof. Unfortunately, most mechanized proofs still spend significant amounts of code on the uninteresting cases, with significant effort expended to write that code. When it comes time to change a theorem statement, say because a language has

been extended with a new construct, the user needs to go back over all of his very manual proofs, editing and adding cases. This is especially tedious with traditional manual proofs in Coq, where proofs are completely unstructured series of commands that modify proof states. Declarative proof languages like Isabelle’s Isar (Wenzel 1999) help alleviate much of this complication, but they do so arguably at the expense of greater verbosity of proofs and greater expenditure in building the first version of a formal development. Is there an even more effective means of structuring proof scripts, such that we can realize evolvability benefits similar to those that software engineers have come to expect?

In the course of this paper, we hope to convince the reader that the answer is “yes.” We will describe our experience building a verified compiler for an impure functional language in Coq. Three main contributions underly the implementation.

- We apply the *parametric higher-order abstract syntax* technique (Chlipala 2008) on a larger, more realistic compiler verification case study than in previous work. This encoding lets us avoid any code dealing with name freshness or index rearrangement in our compiler pass implementations, and that simplicity makes it easier to write correctness proofs.
- To avoid the usual deluge of lemmas about substitution, we use a new approach to encoding operational semantics. Substitution does not appear explicitly and is instead delegated to the meta language, as in the classical HOAS approach, but in a way compatible with general-purpose type theories like Coq’s.
- Each of our Coq proof scripts is a program that is able to adapt to changes to language definitions and theorem statements. Such proof scripts express what are, in our opinion, the real essences of why theorems are true, the insights that could stump someone coming at the proofs from scratch.

Our approach is a synergistic combination of lightweight representations and aggressive theorem-specific automation. We implemented a first version of our compiler for a source language missing several of the features from the final version: let expressions, constants, equality testing, and recursive functions. We were able to add these features after-the-fact with minimal alterations of and additions to our proof scripts; the extended proofs do not even mention the new syntactic constructs or the operational semantics rules that govern them.

Some of the ideas we present here can be mapped back into alternative ways of doing things in pencil-and-paper semantics, but, at the level of detail that is traditional in venues like POPL, our techniques would probably only increase proof length and complexity. Instead, this paper is focused on how to engineer a verified compiler for a functional language. The trickiest parts of doing this with a proof assistant turn out to have little relation to the trickiest parts of doing it on paper. Representations matter a lot, and proof structuring techniques have a serious impact on how easy it is to evolve a verified compiler over time.

Past projects have considered verifying compilers for pure functional languages. As far as we are aware, ours is the first to consider a functional source language with either of mutable references or exceptions. On paper, these features seem straightforward to add to a compiler proof. In a proof assistant, when using the most straightforward proof techniques, impurity infects all of the main theorems and lemmas. It seems a shame to pass up the opportunity to automate the flow of these details through our proofs, and we do our best to take advantage of the possibility.

1.1 The Case Study

Our compiler operates on programs in a kind of untyped Mini-ML, as shown in Figure 1. We have constants from some unspec-

Constants	c
Variables	x, f
Expressions	$e ::= c \mid e = e \mid x \mid e\ e \mid \text{fix } f(x). e$ $\mid \text{let } x = e \text{ in } e$ $\mid () \mid \langle e, e \rangle \mid \text{fst}(e) \mid \text{snd}(e)$ $\mid \text{inl}(e) \mid \text{inr}(e)$ $\mid \text{case } e \text{ of } \text{inl}(x) \Rightarrow e \mid \text{inr}(x) \Rightarrow e$ $\mid \text{ref}(e) \mid !e \mid e := e$ $\mid \text{raise}(e) \mid e \text{ handle } x \Rightarrow e$

Figure 1. Source language syntax

Registers	$r ::= r_0 \mid \dots \mid r_{N-1}$
Constants	$n \in \mathbb{N}$
Lvalues	$L ::= r \mid [r + n] \mid [n]$
Rvalues	$R ::= n \mid r \mid [r + n] \mid [n]$
Instructions	$I ::= L := R \mid r \stackrel{+}{=} n$ $\mid L := R \stackrel{?}{=} R \mid \text{jnz } R, n$
Control-flow instructions	$J ::= \text{halt } R \mid \text{fail } R \mid \text{jmp } R$
Basic blocks	$B ::= (I^*, J)$
Programs	$P ::= (B^*, B)$

Figure 2. Target assembly language syntax

ified base types, comparable with a primitive equality operation; recursive functions; let-binding; unit values; products; sums; mutable references; and exceptions. This language is meant to capture the key features of core ML’s dynamic semantics, omitting essential features only when they involve variable numbers of arguments or variable binding structure. In particular, we do not model variable-arity products and sums, mutually-recursive functions, or compound pattern matching.

Our target language is the idealized assembly language shown in Figure 2. It differs from a real assembly language in representing words with natural numbers and in supporting an infinite memory bank of words. There is still a finite supply of N registers. Our particular compiler works for any $N \geq 3$, allocating some variables to the additional registers when possible. An assembly program consists of a list of basic blocks with one distinguished basic block where execution begins. A basic block is a sequence of instructions terminated by a control-flow instruction. The supported varieties of instruction are assignment using different addressing modes (where $[.]$ operands denote memory accesses), increment of a register by a constant, equality comparison, and conditional jump based on whether a value is nonzero. Control-flow instructions include halt, for normal program termination; fail, for termination on an uncaught exception; and jmp, the standard “computed goto.” Each halt or fail instruction takes an additional program result code as an argument. The destination operands to jnz and jmp are interpreted as indices into the program’s list of basic blocks.

The compiler is idealized in another important way. Unlike in our past work on a compiler for basic lambda calculus (Chlipala 2007), there is no interface with a garbage collector. The output assembly programs allocate new memory but never free any memory. As our present focus is on reasoning about nested binders, we leave the low-level treatment of memory management to future work.

We give the source language a standard big-step operational semantics. Figure 3 shows a sampling of the rules. We define a separate syntactic class of *values* (associated with the metavariable v) in the usual way, taking a restriction of the syntax of expressions and adding a form $\text{ref}(n)$, standing for an allocated reference cell

$(h, \text{fix } f(x). e) \Downarrow (h, \text{Ans}(\text{fix } f(x). e))$
$\frac{(h_1, e_1) \Downarrow (h_2, \text{Ans}(\text{fix } f(x). e)) \quad (h_2, e_2) \Downarrow (h_3, \text{Ans}(e'))}{(h_3, e[f \mapsto \text{fix } f(x). e][x \mapsto e']) \Downarrow (h_4, r)}$
$\frac{}{(h_1, e_1 e_2) \Downarrow (h_4, r)}$
$\frac{(h_1, e_1) \Downarrow (h_2, \text{Ex}(v))}{(h_1, e_1 e_2) \Downarrow (h_2, \text{Ex}(v))}$
$\frac{(h_1, e_1) \Downarrow (h_2, \text{Ans}(\text{fix } f(x). e)) \quad (h_2, e_2) \Downarrow (h_3, \text{Ex}(v))}{(h_1, e_1 e_2) \Downarrow (h_3, \text{Ex}(v))}$
$\frac{(h_1, e) \Downarrow (h_2, \text{Ans}(v))}{(h_1, \text{ref}(e)) \Downarrow (v :: h_2, \text{Ans}(\text{ref}(h_2)))}$
$\frac{(h_1, e) \Downarrow (h_2, \text{Ans}(\text{ref}(n))) \quad h_2.n = v}{(h_1, !e) \Downarrow (h_2, \text{Ans}(v))}$
$\frac{(h_1, e) \Downarrow (h_2, \text{Ans}(v))}{(h_1, \text{raise}(e)) \Downarrow (h_2, \text{Ex}(v))}$
$\frac{(h_1, e_1) \Downarrow (h_2, \text{Ex}(v)) \quad (h_2, e_2[x \mapsto v]) \Downarrow (h_3, r)}{(h_1, e_1 \text{ handle } x \Rightarrow e_2) \Downarrow (h_3, r)}$

Figure 3. Sample rules from source language semantics

at numerical address n . The basic judgment is $(h_1, e) \Downarrow (h_2, r)$, where h_1 and h_2 are reference cell heaps represented as lists of values, e is the expression to evaluate, and r is the result, which is either $\text{Ans}(v)$ for normal termination with expression result v or $\text{Ex}(v)$ when v was raised as an exception and not caught. There are 38 rules in total, covering all of the points in an expression’s evaluation where an exception might be raised. We write $|h|$ for the length of a list h .

Our assembly language also has a big-step operational semantics. There are many equivalent ways of formalizing such a semantics; the specifics that we chose will not matter in what follows. The overall judgment is of the simple form $P \Downarrow h, r$, where h is the final heap and r is either $\text{halt}(n)$ or $\text{fail}(n)$.

Our final theorem says that compiled programs have the same observable behavior as their corresponding source programs. With our limited languages, a program can only exhibit one of three kinds of observable behavior: halting, failing, or diverging. The theorem we prove in this case study ignores non-termination. Our theorem is enough to translate facts about terminating source programs into facts about assembly programs, which is the main kind of verification of interest for a deterministic source language without I/O. We plan eventually to strengthen the final theorem by applying co-inductive big-step operational semantics (Leroy and Grall 2009) to prove that divergent source programs are also mapped to divergent assembly programs.

We write $[e]$ for the compilation of expression e . Stating the final theorem requires formalizing the “contract” between the compiler and the programmer. The compiler writer agrees to follow certain data layout conventions, but it is useful to leave some aspects of representation unspecified, to avoid unnecessary restrictions on

$$\frac{h \vdash \text{fix } f(x). e \cong n \quad h \vdash () \cong n \quad h \vdash \text{ref}(n) \cong n'}{h \vdash c \cong [c]}$$

$$\frac{h \vdash v_1 \cong h[n] \quad h \vdash v_2 \cong h[n+1]}{h \vdash (v_1, v_2) \cong n}$$

$$\frac{h[n] = 0 \quad h \vdash v \cong h[n+1] \quad h[n] = 1 \quad h \vdash v \cong h[n+1]}{h \vdash \text{inl}(v) \cong n \quad h \vdash \text{inr}(v) \cong n}$$

Figure 4. The compiler’s data layout contract

optimization. We chose to give a full specification for all kinds of data besides functions and references. Other decisions are possible, with minimal impact on the proofs. A much more specific relation underlies our main inductive proofs, and we arrive at the visible contract simply by forgetting some details of the “real” relation.

The data layout contract is specified as a relation $h \vdash v \cong n$, parameterized by the final assembly-level heap h and relating source-level value v to assembly-level word n . Figure 4 gives the details. We overload the notation $[c]$ to denote the compilation of a source-level constant into a word. Our development is parameterized over an arbitrary injective function of this kind.

We lift this relation in the natural way to apply to program results.

$$\frac{h \vdash v \cong n}{h \vdash \text{Ans}(v) \cong \text{halt}(n)} \quad \frac{h \vdash v \cong n}{h \vdash \text{Ex}(v) \cong \text{fail}(n)}$$

Now our main theorem can be stated succinctly.

THEOREM 1 (Semantic correctness of compilation). *For any source program e , heap h , and result r , if $(\cdot, e) \Downarrow (h, r)$, then there exist h' and r' such that $[e] \Downarrow h', r'$ and $h' \vdash r \cong r'$.*

1.2 Outline

In the next section, we review the higher-order syntax representation scheme that we introduced in prior work. In Section 3, we present a new substitution-free approach to encoding operational semantics. Section 4 describes, with discussion of their correctness proofs, the main phases of our compiler: conversion from first-order to higher-order syntax, CPS translation, closure conversion, translation to three-address code, and assembly code generation. Section 5 discusses our verified optimizations: common subexpression elimination, dead code elimination, and register allocation. Section 6 gives some statistics about our implementation, Section 7 compares with related work, and we conclude in Section 8.

The case study that we describe in this paper is included in the directory `examples/Untyped` of the latest release of our Lambda Tamer library for compiler verification in Coq, available at

<http://ltamer.sourceforge.net/>

2. Parametric Higher-Order Abstract Syntax

In this section, we summarize key elements of our past work on representing programming language syntax. The following section presents new material that is crucial for scaling up to more realistic languages.

To formalize reasoning about languages with nested variable binders, one needs to settle on a binding representation. Such detail is often swept under the rug in pencil-and-paper proofs. Taking many informal presentations literally, we arrive at concrete representations like the one embodied in this Coq datatype definition for the abstract syntax of untyped lambda calculus.

```

Inductive exp : Type :=
| Var : string -> exp
| App : exp -> exp -> exp
| Abs : string -> exp -> exp.

```

A type definition like this one does not implement usual conventions on its own. At a minimum, we need some well-formedness judgment characterizing when an expression is free of dangling variable references. This means that each of our proofs must include extra premises characterizing which expressions are well-formed with respect to which variable environments. If our formalization requires a notion of capture-avoiding substitution, we need to define one manually. We must also prove a fair number of extra lemmas about well-formedness and substitution. These lemmas must have different proofs for different object languages.

There are other so-called *first-order* representation schemes that alleviate this burden somewhat, including the de Bruijn index, nominal, and locally nameless styles that we mentioned earlier. Significant extra reasoning about freshness and/or well-formedness remains. An alternative is to use *higher-order abstract syntax (HOAS)* (Pfenning and Elliot 1988), which represents object language binders using meta language binders. This pseudo-Coq definition captures the way we revise concrete syntax to arrive at HOAS.

```

Inductive exp : Type :=
| App : exp -> exp -> exp
| Abs : (exp -> exp) -> exp.

```

For instance, we represent an application of the identity function to itself with `App (Abs (fun x => x)) (Abs (fun x => x))`. We encode the matching-up of binders with their uses by borrowing our meta language's facility for that kind of matching with anonymous functions.

We called this definition “pseudo-Coq” because Coq will not accept it. An inductive type is not allowed to be defined with a constructor that takes as input a function over the same type. Allowing this would be problematic because it would allow the coding of non-terminating programs, even without use of explicit recursive definitions, by taking advantage of the opportunity to write “exotic terms” that do not correspond to real lambda terms. Since Coq follows the Curry-Howard Isomorphism in identifying proofs with functional programs, non-termination corresponds to logical inconsistency, where any theorem can be “proved” spuriously with an infinite loop. Systems like Twelf avoid this problem by using weaker meta languages like LF (Harper et al. 1993) that crucially omit features like pattern-matching and recursion.

Even in general-purpose functional programming languages, HOAS terms are difficult to deconstruct manually. It is not generally possible to “go under a binder,” since languages like ML and Haskell provide no way to introspect into closures at runtime. Washburn and Weirich (2008) proposed a technique for fixing some of these deficiencies by taking advantage of parametric polymorphism. Guillemette and Monnier (2008) showed how the technique can be combined with generalized algebraic datatypes to do static verification of compiler type preservation in GHC Haskell.

Washburn and Weirich’s encoding still is not accepted literally by Coq, but a small variation achieves similar benefits. In past work (Chlipala 2008), we showed how to use parametric higher-order abstract syntax (henceforth abbreviated “PHOAS”) to formalize the syntax and semantics of programming languages. We were able to construct very simple, highly-automated proofs of the correctness of some transformations on functional programs.

Here is the syntax of lambda calculus reformulated in PHOAS.

```

Section var.
Variable var : Type.

```

```

Inductive exp : Type :=
| Var : var -> exp
| App : exp -> exp -> exp
| Abs : (var -> exp) -> exp.

```

End var.

```

Definition Exp : Type := forall var : Type, exp var.

```

We use Coq’s section mechanism to scope a local variable over a definition. Outside of the section, the `exp` type becomes a type family parameterized by a choice of `var` type. This definition is accepted by Coq, and the crucial difference from HOAS is that a binder is represented as a function over *variables*, rather than over expressions. This satisfies Coq’s positivity constraint for inductive definitions. Now the identity function can be written as `Abs (fun x => Var x)`, with some choice of `var` fixed globally.

Considering just the part of the above code inside the section, we are using the encoding known as weak HOAS (Honsell et al. 2001). If we treat the type `var` as an unknown, Coq’s type system ensures that every `exp` corresponds to a real lambda term, since it is not possible for a function over variables to do anything interesting based on its input values, which in effect come from an abstract type. The parametric part of PHOAS comes in treating `var` as more than just a global unknown. We define our final expression representation type `Exp` such that an expression is a first-class polymorphic function from a choice of `var` to an `exp` that uses that choice. For instance, the final PHOAS form of the identity function is `fun var => Abs var (fun x : var => Var var x)`. The parametricity of the meta language makes this scheme equivalent to treating `var` as a global constant, but we gain the ability to instantiate `Exps` with particular `var` choices to help us write particular functions.

For instance, we can implement capture-avoiding substitution like this:

```

Section flatten.
Variable var : Type.

```

```

Fixpoint flatten (e : exp (exp var)) : exp var :=
  match e with
    | Var e' => e'
    | App e1 e2 => App (flatten e1) (flatten e2)
    | Abs e1 => Abs (fun x => flatten (e1 (Var x)))
  end.

```

End flatten.

```

Definition Exp1 := forall var : Type, var -> exp var.
Definition Subst (E : Exp1) (E' : Exp) : Exp :=
  fun var => flatten (E (exp var) (E' var)).

```

First, we write a function `flatten` that “flattens” an expression where variables are themselves represented as expressions. Every variable is replaced by the expression that it holds. We recurse inside an `Abs` constructor by building a new argument for `Abs` that itself calls `flatten`. The recursive call is on the original function body applied to a particular locally-bound variable.

We can use `flatten` to implement substitution easily. We define `Exp1`, the PHOAS type of an expression with one free variable. Substitution of `E'` in `E` is implemented with an anonymous polymorphic function over a `var` choice. For a particular `var`, we instantiate the one-hole expression `E` to represent variables as expressions and the substitutand `E'` to represent variables with `var`. Applying the former to the latter, we arrive at an expression whose flattening is the proper result of substitution.

In past work (Chlipala 2008), we showed how to use PHOAS to give denotational semantics to statically-typed, strongly-normalizing functional languages. The basic trick is to parameterize variables by types and define a type denotation function that can be used as a variable representation in implementing the expression denotation function. Using this encoding, we implemented and proved totally correct a number of common functional language compiler passes. These proofs usually rely on the fact that values of types like `Exp` are “really parametric.” We formalized this property in terms of a judgment that axiomatizes equivalence between expressions that use different variable representations. This judgment for untyped lambda calculus is $\Gamma \vdash e_1 \sim e_2$ as defined below. Where e_i represents variables in var_i , Γ is a context of pairs in $\text{var}_1 \times \text{var}_2$. We write $\#x$ for the `Var` constructor applied to x and λf for `Abs` applied to f .

$$\frac{(x_1, x_2) \in \Gamma \quad \Gamma \vdash e_1 \sim e'_1 \quad \Gamma \vdash e_2 \sim e'_2}{\Gamma \vdash \#x_1 \sim \#x_2} \quad \frac{}{\Gamma \vdash e_1 e_2 \sim e'_1 e'_2}$$

$$\frac{\forall x_1, x_2. \Gamma, (x_1, x_2) \vdash f_1(x_1) \sim f_2(x_2)}{\Gamma \vdash \lambda f_1 \sim \lambda f_2}$$

A parametric expression E is well-formed if, for any var_1 and var_2 , we have $\cdot \vdash E(\text{var}_1) \sim E(\text{var}_2)$. We conjecture that this statement holds true for any E of the proper type, and we asserted that fact as an axiom in our past work. In the case study of this paper, we instead prove that expressions are well-formed as needed. Future theoretical work that proved the consistency of this family of axioms would remove the need for specialized well-formedness proofs.

In proving the correctness of a program transformation, it is generally the case that we use one variable choice in evaluating a source program and a different variable choice in translating the program. We use the well-formedness of the expression to derive that the two instantiated expressions are equivalent. Proofs then tend to proceed by induction or inversion on these concrete well-formedness derivations.

At this point, the reader may want to accuse us of misleading advertising, since earlier we complained that first-order representations require too much bookkeeping about well-formedness. The key difference with PHOAS is that (we conjecture) every expression is well-formed by construction. We materialize well-formedness proofs only as needed, and we never need to prove PHOAS analogues of common lemma schemas like substitution, weakening, and permutation. In this case study, we proved well-formedness manually where needed as a kind of due diligence, but we anticipate that the theory will eventually be in place to rest easily assuming axioms of universal well-formedness. In any case, PHOAS avoids the need to generate fresh names or rearrange existing names in implementing a wide variety of transformations. These administrative operations are the bane of programming and proving with first-order representations.

3. Substitution-Free Operational Semantics

Our past work gives programs semantics by interpretation into Coq’s strongly-normalizing logic CIC; thus, that work cannot be applied directly to Turing-complete programming languages like our source and target languages here. The main representational innovation of our new work is an effective way of writing operational semantics over PHOAS terms. Operational semantics has proved its worth in the formalization of a wide variety of languages, so our new encoding expands the effective range of PHOAS dramatically.

It is tempting to write operational semantics directly over parametric terms (e.g., in `Exp` from the last section). Doing so is actually fairly straightforward, with the trick for implementing substitution

as the one surprise. Unfortunately, inductive proofs over parametric terms tend to involve just as much administrative overhead as we find with first-order representations. Dealing with instantiated terms (e.g., in `exp`) frees us to leave variables deep within syntax trees annotated with arbitrary meta language values. If we work with parametric terms, we must instead represent and apply contexts explicitly in our induction hypotheses, since it is impossible to “go under a binder” without first fixing a `var` choice.

Moreover, working with substitution explicitly brings back the same family of lemmas about the interaction of substitution and other functions. Generally we must prove at least one lemma about substitution for each program transformation function that we write. The details of such lemmas are almost always elided in pencil-and-paper proofs, but we must prove them in full detail to satisfy a proof assistant.

The alternative that we use here is to define an operational semantics over instantiated terms that avoids mentioning substitution explicitly. Our encoding has the flavor of a hybrid between a high-level semantics and an abstract machine, where we track closure allocation explicitly. We want to write semantics equivalent to examples like the one in Figure 3. For basic lambda calculus, it is tempting to start out by defining a type `val` of values like the following, such that we can instantiate `var` as `val` in our semantics.

```
Inductive val : Type :=
| VAbs : (val -> exp val) -> val.
```

This definition suffers from the same problem as our earlier HOAS pseudo-definition: we try to define `val` in terms of functions over itself. Coq rejects the definition as ill-formed, which is a good thing, because otherwise we would be able to implement a lambda calculus interpreter in Coq, which gives us a trivial way of coding an infinite loop and thus breaking logical consistency.

Our solution is to represent values as natural numbers that index into a heap of “closures,” or meta language functions from values to expressions. The technique bears a resemblance to approaches to making allocation explicit in operational semantics, e.g., as in Morrisett et al. (1995). That line of work aims to capture how high-level programs execute on real machines, while keeping at the right level of abstraction. In contrast, our use of explicit allocation is aimed at removing the need to reason about explicit substitution. What we have here is really just an instance of a common pattern in semantics of moving to more explicitly syntactic techniques to circumvent circularities in type definitions.

```
Definition val : Type := nat.
Definition closure : Type := val -> exp val.
Definition closures : Type := list closure.
```

Here is a big-step semantics in this style for basic lambda calculus. Its Coq type signature could be `closures -> exp val -> closures -> val -> Prop`.

$$\frac{(H, \#v) \Downarrow (H, v) \quad (H, \lambda f) \Downarrow (f :: H, |H|)}{(H_1, e_1) \Downarrow (H_2, n) \quad (H_2, e_2) \Downarrow (H_3, v) \quad H_3.n = f \quad (H_3, f(v)) \Downarrow (H_4, v')}$$

$$(H_1, e_1 e_2) \Downarrow (H_4, v')$$

As with HOAS and its relatives in general, we manage to delegate the object-language-specific handling of substitution to the meta language. This delegation happens in the occurrence of $f(v)$ in the application rule. The rule for variables is also more interesting than it may appear at first. By evaluating a variable node $\#v$ to its content v , we effectively push the operation of last section’s `flatten` function into our semantics.

The trickiness of usual substitution stems from the need to reason about nested binder scopes. We have replaced that kind of

reasoning with global reasoning about a closure heap. We can prove a relatively small set of lemmas about lists and reuse it to handle closure heaps in all developments that use our encoding. There is no need to prove even a single lemma about substitution upon starting with a new object language or transformation.

Our technique generalizes to the full source language from Figure 1. We revise our `val` definition like this, using the illustrative type synonym `label` for `nat` from our library:

```
Inductive val : Type :=
| VFunc : label -> val
| VUnit : val
| VPair : val -> val -> val
| VInl : val -> val
| VInr : val -> val
| VRef : label -> val.
```

The main PHOAS semantics for the source language tracks input and output versions of both a closure heap and a reference heap. We reuse our library of list lemmas to reason about both kinds of heaps.

The definitions above are about expressions specialized to represent variables as values, but it is now easy to define an evaluation relation for parametric terms.

$$\frac{(\cdot, E(\text{val})) \Downarrow (H', v)}{E \Downarrow v}$$

We have modified standard operational semantics by adding a level of indirection. In a traditional paper proof at the traditional level of detail, our change would only add bureaucratic hassle. Counterintuitively, the change *reduces* hassle in mechanized proofs, since it helps us delegate to the meta language some details of processing the object language.

The substitution-free encoding is more than just a trick to place PHOAS on a level playing field with first-order representations. PHOAS with our new semantic encoding compares very favorably with other known combinations of syntactic and semantic encodings. As far as we are aware, every competing technique is either invalid in a general-purpose proof system or leads to proof overhead significantly above that in our implementation. In verifying all of our translations that represent syntax solely with PHOAS, there is not a single lemma establishing one of the standard object-language-specific syntactic properties. We only once use proof by induction over the structure of programs, and that occurs for an auxiliary lemma relevant to closure conversion, where explicit reasoning about variables is hard to avoid.

Substitution-free operational semantics has much in common with environment semantics, where an evaluation judgment takes an additional input which assigns a value to each free variable of the expression to evaluate. Both approaches involve explicit first-order treatment of an aspect of evaluation that is implicit in the more common varieties of natural semantics. Where environment semantics treats every variable in a first-order way, substitution-free semantics does the same with closures. We have found the latter to have serious advantages for proof engineering. None of our translations includes more than one case that has any interesting effect on closure allocation sequence. As a result, none of our proofs includes more than one case that must compensate for the effect of such a change on semantics. In contrast, with environment semantics, almost every case of each of our translations would need some accounting for a rearrangement of variable binding structure.

Theorem-specific proof automation is one of the core techniques in our approach to compiler verification, and we will have more to say later on the details of that automation. While first-order encodings of operational semantics usually mention substitution

explicitly, the standard lemmas about substitution tend to admit simple automated proof strategies. Nonetheless, we feel it is still a significant burden to have to state and apply all of these lemmas explicitly. Perhaps an automation package could go further and apply substitution properties automatically as needed. In the project described in this paper, we have avoided any automatic application of induction, which rules out proofs of the usual syntactic lemmas. We view our more elementary automation style as a mark in favor of our proposal.

More standard operational semantics with explicit substitution is easier to understand and believe, so we chose to state the final theorem independently of the new technique. The first part of the next section shows how we convert from first-order to higher-order syntax and semantics, along with how we justify the soundness of the conversion.

4. Main Compiler Phases

In this section, we walk through our compiler intermediate languages and the phases that translate into them. We will overload notation by writing $[\cdot]$ for the compilation function being discussed in each subsection.

For lack of space, we discuss our proof automation techniques only in the context of CPS translation, in Section 4.2.2. The design patterns introduced there apply equally well to the other translations.

4.1 PHOASification

Our final theorem is stated entirely in terms of standard encodings of syntax and semantics, with no mention of PHOAS. We achieve this by beginning our compiler with a de Bruijn index (de Bruijn 1972) implementation of the syntax from Figure 1. The first compiler phase translates these programs into PHOAS equivalents, where the PHOAS syntax is a different encoding of Figure 1.

It is not generally possible to “cheat” in implementing a translation into PHOAS, in the sense that there is no default value to use for variables when it turns out that the input program is ill-formed somehow. Therefore, we use dependent types to enforce that every source program is closed by construction. This is a standard technique in the dependent types world, where a type `exp` is indexed by a natural number expressing how many free variables are available. Variables in ASTs are represented in types `fin n`, which are isomorphic to sets of natural numbers below n .

Our implementation of PHOASification uses another standard dependent type family, which we call `ilist` in our library. For a type `T` and a natural number n , an `ilist T n` value is a length- n list of values in `T`.

The type of the main translation is `forall (var : Type) (n : nat), Source.exp n -> ilist var n -> Core.exp var`. Besides the expression to translate, $[\cdot]$ takes in a list representing a mapping from de Bruijn indices to PHOAS variables. Here are some representative cases from the function’s definition, where we write $\sigma.f$ for the projection from the `ilist` σ of the value at the position indicated by `fin` value f . We write $\hat{\lambda}$ for the meta language’s function abstraction.

$$\begin{aligned} [\#x] \sigma &= \#(\sigma.x) \\ [e_1 e_2] \sigma &= [e_1] \sigma [e_2] \sigma \\ [\text{fix } f(x). e_1] \sigma &= \text{fix}(\hat{\lambda} f. \hat{\lambda} x. [e_1] (x :: f :: \sigma)) \end{aligned}$$

In the source language definition, we simplify the definition of substitution by defining a type of values and including an `exp` constructor that allows the injection of any value into any type `exp n`. Thus, the closed nature of a value is apparent from its type, so we avoid needing to lift de Bruijn indices in the process of substituting a value in an open term.

4.1.1 Correctness Proof

We define two mutually-inductive relations, for characterizing the compatibility of source and PHOAS expressions and values. Both relations are parameterized by PHOAS-level closure heaps, and the expression relation is also parameterized by an `ilist`, like in the definition of the translation. Here are a few representative cases. We write $\$$ for the constructor that injects source values into source expressions, and we write $H \rightsquigarrow H'$ for the fact that H is a suffix of H' . We use the notation $\text{fix } F$ for PHOAS-level application of the AST constructor for recursive functions.

$$\begin{array}{c} \frac{H \vdash \sigma.f \simeq v \quad H, \sigma \vdash e_1 \simeq e'_1 \quad H, \sigma \vdash e_2 \simeq e'_2}{H, \sigma \vdash \#f \simeq \#v \quad H, \sigma \vdash e_1 e_2 \simeq e'_1 e'_2} \\ \frac{\forall f, x. H, x :: f :: \sigma \vdash e \simeq F(f)(x) \quad H \vdash v \simeq v'}{H, \sigma \vdash \text{fix } f(x). e \simeq \text{fix } F \quad H, \sigma \vdash \$v \simeq \#v'} \\ \frac{H.n = f \quad \forall x_1, x_2, H'. H \rightsquigarrow H' \Rightarrow H', x_2 :: x_1 :: \vdash e \simeq f(x_1)(x_2)}{H \vdash \text{Fix}(e) \simeq \text{Fix}(n)} \\ \frac{H \vdash v_1 \simeq v'_1 \quad H \vdash v_2 \simeq v'_2}{H \vdash \text{Pair}(v_1, v_2) \simeq \text{Pair}(v'_1, v'_2)} \end{array}$$

We prove that both relations are monotone, with respect to replacing a heap H with another heap H' such that $H \rightsquigarrow H'$. We also lift the value relation to apply over results $\text{Ans}(\cdot)$ and $\text{Ex}(\cdot)$ in the natural way.

There are two main lemmas behind the correctness theorem for this phase. First, we prove that the compilation function respects expression compatibility.

LEMMA 1. *For any e and σ with compatible type indices, if e contains no uses of $\$$, then $\cdot, \sigma \vdash e \simeq [e] \sigma$.*

From this starting point, we track the parallel execution of e and its compilation. We use a notion of reference heap compatibility $H \vdash h \simeq h'$, which says that h and h' have the same length and that their values belong pairwise to $H \vdash \cdot \simeq \cdot$.

LEMMA 2. If:

- $(h_1, e) \Downarrow (h_2, r)$ at the source level,
- **And** $H, \sigma \vdash e \simeq e'$,
- **And** $H \vdash h_1 \simeq h'_1$,

Then there exist H' , h'_2 , and r' such that:

- $(H, h'_1, e') \Downarrow (H', h'_2, r')$,
- **And** $H' \vdash r \simeq r'$,
- **And** $H' \vdash h_2 \simeq h'_2$.

Lemma 2 appeals to an auxiliary lemma about substitution. We note that this is the only place in our development where a substitution theorem is proved explicitly.

These lemmas together yield the final theorem directly.

THEOREM 2. *If $(\cdot, e) \Downarrow (h, r)$ and e is closed and does not use $\$$, then there exist H , h' , and r' such that $(\cdot, \cdot, [e] \cdot) \Downarrow (H, h', r')$ and $H \vdash r \simeq r'$.*

4.2 Conversion to Continuation-Passing Style

The first main compiler phase translates programs into continuation-passing style. Functions no longer return, explicit exception handling constructs are eliminated, and expression evaluation is broken up with sequences of let bindings of the results of primitive operations on variables. Figure 5 shows the syntax of the translation's target language.

$$\begin{array}{ll} \text{Primops} & p ::= c \mid x = x \mid \text{fix } f(x). e \\ & \mid () \mid \langle x, x \rangle \mid \text{fst}(x) \mid \text{snd}(x) \\ & \mid \text{inl}(x) \mid \text{inr}(x) \mid \text{ref}(x) \mid !x \mid x := x \\ \text{Expressions} & e ::= \text{halt}(x) \mid \text{fail}(x) \mid x \ x \mid \text{let } x = p \text{ in } e \\ & \mid \text{case } x \text{ of } \text{inl}(x) \Rightarrow e \mid \text{inr}(x) \Rightarrow e \end{array}$$

Figure 5. CPS language syntax

$$\begin{array}{ll} \lfloor \#x \rfloor k_{SK_E} & = k_S(x) \\ \lfloor \text{raise}(e) \rfloor k_{SK_E} & = x \xleftarrow{k_E} e; k_E(x) \\ \lfloor \text{let } x = e_1 \text{ in } e_2 \rfloor k_{SK_E} & = x \xleftarrow{k_E} e_1; \lfloor e_2 \rfloor k_{SK_E} \\ \lfloor e_1 \ e_2 \rfloor k_{SK_E} & = f \xleftarrow{k_E} e_1; x \xleftarrow{k_E} e_2; \\ & \text{let } k'_S = \lambda r. k_S(r) \text{ in} \\ & \text{let } k'_E = \lambda r. k_E(r) \text{ in} \\ & \text{let } p' = \langle k'_S, k'_E \rangle \text{ in} \\ & \text{let } p = \langle x, p' \rangle \text{ in } f \ p \end{array}$$

Figure 6. CPS translation

We use a higher-order one-pass CPS translation, in the style of Danvy and Filinski (1992). The type of the translation is `forall var, Core.exp var -> (var -> CPS.exp var) -> (var -> CPS.exp var) -> CPS.exp var`. Beyond the input expression, the extra arguments are the current success continuation and the current exception handler, represented as meta language functions over result variables. Figure 6 shows some representative cases of the definition. We write $\lambda x. e$ as shorthand for $\text{fix } f(x). e$ when f does not occur free in e . We write $x \xleftarrow{k_E} e_1; e_2$ as shorthand for $\lfloor e_1 \rfloor (\hat{\lambda} x. e_2)(k_E)$. The application case demonstrates how the effective domain of each core function is expanded to 3-tuples of a main argument, a success continuation, and an exception handler.

This compilation function takes as an argument a choice of `var` representation. In compiling a parametric expression E , we return an abstraction over `var`, within which we call the concrete compilation function with `var` and $E(\text{var})$ as arguments. We pass always-halt and always-fail functions as the success continuation and exception handler, respectively.

4.2.1 Correctness Proof

As for the last phase, this correctness proof is based around a relation between core and CPS values. Since both languages use PHOAS, the relation is parameterized by a closure heap for each. Here are some representative rules. For a meta language function f representing a function abstraction body, we write $\lfloor f \rfloor$ for the way that the main compilation translates that body. The relation definition below depends on a version of the \sim relation from Section 2, extended to apply to the input language.

$$\begin{array}{c} H.n = f \quad H'.n' = \lfloor f' \rfloor \\ (\forall x_1, x'_1, x_2, x'_2. \quad \Gamma, (x_1, x'_1), (x_2, x'_2) \\ \quad \vdash f(x_1)(x_2) \sim f'(x'_1)(x'_2)) \\ (\forall v, v'. (v, v') \in \Gamma \Rightarrow H, H' \vdash v \simeq v') \\ \hline H, H' \vdash \text{Fix}(n) \simeq \text{Fix}(n') \\ \hline \frac{H, H' \vdash v_1 \simeq v'_1 \quad H, H' \vdash v_2 \simeq v'_2}{H, H' \vdash \text{Pair}(v_1, v_2) \simeq \text{Pair}(v'_1, v'_2)} \quad \frac{}{H, H' \vdash \text{Ref}(n) \simeq \text{Ref}(n)} \end{array}$$

As in the last subsection, we lift this relation in the natural way to pairs of reference heaps and pairs of results.

Our main theorem is with respect to a substitution-free big-step semantics for CPS programs. The signature is the same as for Core but with final heaps no longer specified, since the final result is all that we care about.

THEOREM 3. If:

- $(H_1, h_1, e) \Downarrow (H_2, h_2, r)$ at the source level,
- **And** $\Gamma \vdash e \sim e'$,
- **And** $H_1, H'_1 \vdash h_1 \simeq h'_1$,
- **And**, for every $(v, v') \in \Gamma$, $H_1, H'_1 \vdash v \simeq v'$,

Then for every pair of continuations k_S and k_E , **there exist** H'_2 , h'_2 , and r' such that:

- **If** $r' = \text{Ans}(v)$ and $(H'_2, h'_2, k_S(v)) \Downarrow r''$,
 - **then** $(H'_1, h'_1, [e'] k_{SK_E}) \Downarrow r''$,
- **And, if** $r' = \text{Ex}(v)$ and $(H'_2, h'_2, k_E(v)) \Downarrow r''$,
 - **then** $(H'_1, h'_1, [e'] k_{SE}) \Downarrow r''$,
- **And** $H'_1 \sim H'_2$,
- **And** $H_2, H'_2 \vdash r \simeq r'$,
- **And** $H_2, H'_2 \vdash h_2 \simeq h'_2$.

This theorem about expressions specialized to values-as-variables makes it easy to derive the theorem about parametric expressions when we substitute the initial success and exception continuations for k_S and k_E .

4.2.2 Automating the Proofs

We begin by proving that each of our compatibility relations is monotone with respect to extension of closure heaps, which follows by induction on derivations. We also give one-liner proofs for five more lemmas that massage “obvious” facts into forms that Coq’s automated resolution prover will be able to use. At that point, we are ready to tackle the proof of Theorem 3, which proceeds by induction on core evaluation judgments.

Figure 7 gives the complete proof script for this theorem, implementing in Coq’s domain-specific tactic language Ltac (Delahaye 2000). We will step through the different elements, remarking as appropriate on the design patterns they embody.

The script begins with hints, which extend Coq’s resolution prover, supporting higher-order logic programming in the tradition of Prolog. The first hint suggests that proof search should try each of the rules of the evaluation judgment for CPS-level primops. In this way, we avoid having to mention the rules explicitly, which makes it possible for the proof to keep working even after we add new kinds of primops.

Hint Constructors CPS.evalP.

Next, we use the Hint Resolve command to suggest some other rules and lemmas to be applied automatically during resolution.

Hint Resolve answer_Ans answer_Ex

We use a Hint Extern command to specify a free-form proof search step. We give 1 as an estimate of the cost of this rule, which effects the order in which rules are attempted. After that, we write a pattern to match against a goal. When the goal matches the pattern, we suggest running the proof script to the right of the arrow. In this case, our script suggests unfolding some definitions, so that we expose the syntactic structure of an expression to evaluate, making it clear which operational semantics rules apply.

Hint Extern 1 (CPS.eval _ _ (cpsFunc _ _) _) => unfold cpsFunc, cpsFunc'.

Now we are ready for the main body of the proof. We proceed by induction on the first hypothesis $(H_1, h_1, e) \Downarrow (H_2, h_2, r)$, and we chain onto our use of induction a script to apply to every inductive case. The semicolon operator accomplishes this chaining, and we wrap the per-case script with abstract to prove every case as a separate lemma, which saves memory by freeing some temporary data structures after each lemma.

induction 1; abstract (...)

We begin every case with an inversion on the expression equivalence judgment $\Gamma \vdash e \sim e'$, which is the first remaining hypothesis.
inversion 1;

Next, we call a generic simplification tactic from our Lambda Tamer library. This tactic knows nothing about any particular object language. It relies on a number of built-in Coq automation tactics and adds some new strategies, combining propositional simplification, partial evaluation, resolution proving, rewriting, and common rules for simplifying sets of hypotheses dealing with standard datatypes like natural numbers, lists, and optional values.

simpler;

At this point, the top-level structure of the expressions appearing in a case is known, and we have gotten as far as we can with generic simplification. The next step is to begin a loop over some theorem-specific simplification strategies. Like other tactic-based proof assistants, Coq supports a number of *tacticals*, a kind of higher-order combinators for assembling new proof strategies. We use the repeat tactical to structure our loop. The argument to repeat is a tactic to attempt repeatedly until it no longer applies. Our argument here uses a match tactic expression, which generalizes normal pattern matching in the tradition of ML and Haskell. A match tactic matches on the form of a proof goal, including both hypotheses and conclusion.

repeat (match goal with

Our heuristics are pattern-matching rules, where each pattern has the form HYPS |- CONC. The HYPS section describes conditions on hypotheses and CONC gives a pattern to match against the conclusion to be proved. The former section is a comma-separated list of zero or more entries of the form $H : p$, asserting that there must exist some hypothesis matching pattern p and to whose name the local variable H should be bound.

The first heuristic looks for a hypothesis asserting some fact $H, H' \vdash v_1 \simeq v_2$. In our implementation, such a fact is written as $H & H' |-- v_1 \sim v_2$, using an ASCII notation that we register as a Coq syntax extension or “macro.” Thus, the first pattern matches any goal with a hypothesis over this judgment. For each such hypothesis, we apply the tactic invert_1_2 from our library. This tactic performs inversion if and only if it is possible to deduce from the form of the hypothesis that at most two distinct rules of the underlying judgment could apply. If more than two rules are possible, the tactic invocation fails, triggering backtracking to try a different choice of H or, if that fails, the next rule in our match expression. By using invert_1_2, we avoid having to specialize this heuristic to the details of our object language, for instance by writing one heuristic per case where we can deduce that a particular rule of $\cdot, \cdot \vdash \cdot \simeq \cdot$ must have been used to conclude H .

| [H : _ & _ |-- _ \sim _ | - _] => invert_1_2 H

The next heuristic follows the logic of the previous one, but for the judgment for result compatibility instead of value compatibility, which has a different notation.

| [H : _ & _ |--- _ \sim _ | - _] => invert_1 H

```

Hint Constructors CPS.evalP.
Hint Resolve answer_Ans answer_Ex CPS.EvalCaseL CPS.EvalCaseR EquivRef'.
Hint Extern 1 (CPS.eval _ _ (cpsFunc _ _) _) => unfold cpsFunc, cpsFunc'.

induction 1; abstract (inversion 1; simpler;
repeat (match goal with
| [ H : _ & _ |-- _ `` _ |- _ ] => invert_1_2 H
| [ H : _ & _ |--- _ `` _ |- _ ] => invert_1 H
| [ H : forall G e2, Core.exp_equiv G ?E e2 -> _ |- _ ] =>
  match goal with
  | [ _ : Core.eval ?S _ E _ _ _ ,
    _ : Core.eval _ _ ?E' ?S _ _ ,
    _ : forall G e2, Core.exp_equiv G ?E' e2 -> _ |- _ ] => fail 1
  | _ => match goal with
    | [ k : val -> expV,
      ke : val -> exp val,
      _ : _ & ?s |-- _ `` _ ,
      _ : context [VCont] |- _ ] =>
      guessWith ((fun (_ : val) x => ke x) :: (fun (_ : val) x => k x) :: s) H
    | _ => guess H
  end
end
end; simpler);
try (match goal with
| [ H1 : _, H2 : _ |- _ ] => generalize (sall_grab H1 H2)
end; simpler);
splitter; eauto 9 with cps_eval; intros;
try match goal with
| [ H : _ & _ |--- _ `` ?r |- answer ?r _ _ ] => inverter H; simpler; eauto 9 with cps_eval
end).

```

Figure 7. Complete proof script for Theorem 3

The next and final heuristic from our main loop chooses when and how to apply induction hypotheses (IHes). The first step in that direction is to identify some hypothesis H that has the right syntactic structure to be an IH. Our Coq development uses the predicate `Core.exp_equiv` for the judgment we write as $\Gamma \vdash e \sim e'$ in the statement of Theorem 3, and the code `?E` denotes a pattern variable.

```
| [ H : forall G e2,
  Core.exp_equiv G ?E e2 -> _ |- _ ] =>
```

It would not be effective to apply the IHes in an arbitrary order. Because of the form of Theorem 3, each successful application yields an existentially-quantified conclusion, and eliminating those quantifiers gives us new variables to work with. Those new variables might be needed to instantiate the *universal* quantifiers of a different IH. It turns out that a simple heuristic lets us choose the right IH order in every case: as each IH is associated with an expression, follow the order in which those expressions were evaluated in the original program.

We can track “evaluation order” by inspecting the closure heaps that are threaded through evaluation. One evaluation “comes after” another if the former’s starting closure heap equals the latter’s ending heap. By clearing each IH as we use it, we make it possible to use the following pattern match to identify an IH that is “not ready yet.” In particular, where the current H is for some expression E , there must exist another IH about some expression E' , such that evaluation of E begins where evaluation of E' leaves off, in terms of the flow of a closure heap S . Thus, since E comes after E' , and since we have not yet applied the IH for E' , we are not yet ready to apply the IH for E . We use the `fail` tactic to backtrack to making a different choice of H .

```
match goal with
| [ _ : Core.eval ?S _ E _ _ _ ,
  _ : Core.eval _ _ ?E' ?S _ _ ,
  _ : forall G e2, Core.exp_equiv G ?E' e2 -> _ |- _ ] => fail 1
```

If the last rule finds no matches, then we know that H is the appropriate IH to apply now. We perform a further pattern-match to determine whether we need to apply an instantiation strategy specific to the case of function application, one of the few interesting cases of the translation. Since the general case is simpler, we will discuss it first. We simply apply the tactic `guess`, which comes from our Lambda Tamer library, to our IH. This generates a new *unification variable* for every universal quantifier in the statement of H . Additionally, we apply automatic resolution proving to discharge each hypothesis of H , in the process learning the values of most of the unification variables that we just introduced. For this theorem, unification variables remain for the continuation variables k_S and k_E , but the other unification variables are determined immediately from context. By relying on the versatile `guess` tactic, we avoid almost all object-language-specific application of IHes. This is one of the key techniques supporting proof reuse.

```
| _ => guess H
```

In most proofs, `guess` can handle the “uninteresting” cases that would not be written out in detail in a pencil-and-paper proof. Often a bit more help from the human prover is needed for the cases that lie at the heart of a transformation’s purpose. For CPS translation, the only such case is function application, and we use pattern matching to identify that case and treat it specially. We use the pattern `context [VCont]` to require that some hypothesis

mentions a continuation value, which turns out to be enough to isolate the case of interest. We also bind local names for the success continuation `k` and the exception handler `ke`. Finally, we pattern-match out the only closure heap `s` that is mentioned in a value compatibility hypothesis.

From these variables, we can construct our piece of advice to `guess`. More specifically, we use the variant `guessWith`, which lets us suggest a value to be used to instantiate any universal quantifier of proper type, such that the remaining quantifiers are still instantiated with fresh unification variables. We know that each function call allocates a new continuation each for the success continuation and exception handler. The argument we pass to `guessWith` reflects that knowledge, suggesting a closure heap that has the two new continuations pushed on.

```
match goal with
| [ k : val -> expV,
  ke : val -> exp val,
  _ : _ & ?s |-- _ ~~ _,
  _ : context[VCont] |- _ ] =>
  guessWith ((fun (_ : val) x => ke x)
  :: (fun (_ : val) x => k x) :: s) H
```

Each iteration of the main loop ends with a call to `simpler`, which will take the existentially-quantified conjunctions produced by `guess` and replace them with individual hypotheses that use fresh top-level variables.

After we finish this main loop of heuristics, most of the work in the proof is done. Simple resolution proving can handle most of the remaining goals. We use one additional pattern match to catch a case where it would be useful to add a new hypothesis justified by the library theorem `sall_grab` about heap well-formedness.

```
try (match goal with
| [ H1 : _, H2 : _ |- _ ] =>
  generalize (sall_grab H1 H2)
end; simpler);
```

After that, we call the tactic `splitter` to turn a goal like $\exists x_1, \dots, x_n. \phi_1 \wedge \dots \wedge \phi_m$ into separate goals ϕ_1, \dots, ϕ_m , with each x_i replaced by a fresh unification variable. We solve most of these goals with a call to the resolution prover `eauto`, specifying a proof tree depth of 9 and an additional hint database `cps_eval`, which includes a rule to apply as many CPS operational semantics rules as possible, counted as a single proof step.

```
splitter; eauto 9 with cps_eval;
```

Each remaining goal is solved by case analysis on whether an unknown evaluation result `r` is normal or represents an uncaught exception. More specifically, we find a hypothesis stating a result compatibility fact, we perform inversion on that hypothesis, and we finish off the resulting cases with standard tactics.

```
try match goal with
| [ H : _ & _ |--- _ ~~ ?r
  |- answer ?r _ _ ] =>
  inverter H; simpler; eauto 9 with cps_eval
end).
```

Our inductive proof has 38 cases to consider, with one for each semantic rule. Many of these cases need this kind of further split on results of sub-evaluations. By using automation to structure our proof script, we shield the human proof architect from the need to consider these many cases individually.

4.3 Closure Conversion

The next compiler phase combines the traditional transformations of closure conversion, which changes all functions to take their

Primops	p	...as in last language, minus fix...
Expressions	e	...as in last language...
Programs	$P ::= e \mid \text{let } f = \text{fix } f(x). e \text{ in } P$	

Figure 8. Closed language syntax

free variables as explicit arguments; and hoisting, which moves all function definitions to the top level of a program. Since we are using PHOAS, it is easiest to combine these phases into one, such that the closed nature of function definitions can be apparent syntactically from the fact that they only appear at the top level of a program. Figure 8 shows the syntax of this translation’s target language.

As in our past work on closure conversion with PHOAS (Chilipala 2008), this phase is interesting because we implement it by converting higher-order syntax to first-order syntax, which is passed to a translation that again produces higher-order syntax. Given a parametric expression E , we instantiate it like $E(\text{nat})$, choosing to represent variables as natural numbers. We also follow a specific convention in how we use such a term, which has type $\text{CPS}.\text{exp nat}$. Our convention is isomorphic to the technique of de Bruijn levels, where bound variables that are not inside nested scopes have level 0, the next binders inside these have level 1, and so on. Compared to the more common de Bruijn indices, this technique has the advantage that all occurrences of a given binder’s variable use the same level. Therefore, since PHOAS binders are represented as functions, we can descend into a binder simply by calling its function with the appropriate number.

We formalize this convention with a well-formedness judgment over $\text{CPS}.\text{exp nat}$. Here are the key rules in a restriction to untyped lambda calculus.

$$\frac{f < n \quad x < n \quad n + 1 \vdash f(n) \text{ wf}}{n \vdash f x \text{ wf}} \quad \frac{}{n \vdash \lambda f \text{ wf}}$$

Our closure conversion is dependently-typed, such that it takes a well-formedness proof as input. We prove a theorem saying that, for any well-formed E , we have $0 \vdash E(\text{nat}) \text{ wf}$; and we pass an invocation of this theorem in the initial call to the translation function.

Here is the type of the main translation.

```
forall (var : Type) (n : nat) (e : exp nat), wf n e
  -> (((env var (freeVars n e) -> Closed.exp var)
    -> Closed.prog var)
  -> Closed.prog var)
```

The function `freeVars` calculates the free variable set of an expression. When called like `freeVars n e`, it returns a length- n list of booleans, indicating which variables up to n appear free in e . The type family `env` is parameterized by such a list. An `env var fvs` is a tuple of one `var` variable for each entry of `fvs` that is true.

The main complexity in the translation type comes from a continuation argument. In translating an expression, the form of that expression implies some top-level function definitions that we should add. It is critical that none of these definitions mentions any local variables, or else we would have an ill-formed program. Thus, in translating an expression, we bind its functions and then call a continuation which may bind additional functions. That continuation then, inside its new definitions, calls a sub-continuation with an environment giving values to all free variables.

We hope that a few representative examples, as shown in Figure 9, make the protocol clear. We write `wf` arguments as ϕ . We rely

$\lfloor \text{halt}(x) \rfloor n\phi k$	$= k(\hat{\lambda}\sigma.\text{halt}(\text{get } x \sigma \phi))$
$\lfloor f \ x \rfloor n\phi k$	$= k(\hat{\lambda}\sigma.\text{let } \sigma' = \text{fst}(\text{get } f (\Pi_1(\sigma)) (\pi_1(\phi))) \text{ in}$ $\text{let } f' = \text{snd}(\text{get } f (\Pi_1(\sigma)) (\pi_1(\phi))) \text{ in}$ $\text{let } p = \langle \sigma', \text{get } x (\Pi_2(\sigma)) (\pi_2(\phi)) \rangle \text{ in}$ $f' \ p)$
$\lfloor \text{let } p \text{ in } f \rfloor n\phi k$	$= \lfloor p \rfloor n(\pi_1(\phi))(\hat{k}_P.$ $\lfloor f(n) \rfloor (n+1)(\pi_2(\phi))(\hat{k}_E.k(\hat{\lambda}\sigma.$ $k_P (\Pi_1(\sigma)) (\hat{\lambda}x.$ $k_E (x :: \Pi_2(\sigma))))$)

Figure 9. Representative cases of closure conversion

on a similar primop translation function whose exact type and definition we will not discuss further here.

There are a few unusual things going on in Figure 9. First, we manipulate well-formedness proofs ϕ as data. When we know from the structure of e that a proof ϕ must be deduced from a rule with two premises, we write $\pi_i(\phi)$ for the extraction of the i th premise’s proof. For a function call $f \ x$, the two premises say that the two variables f and x are both less than the current de Bruijn level n . Such less-than proofs may be passed to the get function to enable extracting variables from an environment σ .

We must also do similar splitting of environments. Several cases of the definition of `freeVars` are implemented by joining sets of free variables. When an environment σ has a type based on the union of two sets, then the projections Π_1 and Π_2 translate into environments for those sets. This is always possible to do, since a union of two sets contains any binding required by either set alone.

The case for fix $f(x)$, e , omitted above, is where new function bindings are created. It relies on two auxiliary functions for packing environments into tuples at closure formation sites and unpacking those tuples in function prologues. Free variable information is used to choose which variables to pack.

4.3.1 Correctness Proof

Reasoning about layers of nested continuations is tricky, so we prove the correctness of our translation by defining an alternate translation that does not use continuations. The alternate translation is specialized to the operational semantics of closed programs, where, as in the definition of `val` from Section 3, we represent function values with natural numbers pointing into a closure heap. By specializing the translation to the semantics, we can refer directly to closures and closure heaps, since function addresses are generated in a predictable way.

Here is the type of the alternate translation:

```
forall (n : nat) (e : CPS.exp nat), Closed.closures
-> wf n e -> Closed.closures
  * (env Closed.val (freeVars n e)
    -> Closed.exp Closed.val)
```

A call to this function takes a current closure heap as input, and return values are pairs of extended closure heaps and functions from local variable environments to expressions. We did not just use this version as our initial translation because it works with a program representation where types alone do not guarantee well-formedness; any variable might include an “out-of-bounds” function reference.

It is fairly straightforward to prove a correctness theorem for the alternate translation. We need to prove about two dozen lemmas about operations on free variables, environments, and closure

heaps. As for the last two phases, we define compatibility relations over the values and reference heaps of the source and target languages. With these pieces in place, we can prove the overall correctness theorem with the usual induction on source evaluation derivations.

A final lemma connects the two translations, proved by mutual structural induction over CPS expressions and primops.

4.4 Flattening

After closure conversion, programs are already almost in the form of three-address code, the family of traditional low-level compiler intermediate languages. Beyond the use of structured control flow with `case` expressions, the only serious difference is that closed programs use let-binding of immutable variables instead of manipulation of mutable temporaries, of which each procedure in three-address code has a fixed set.

We make this connection precise by defining a flat language and a translation into it. Every let-bound variable in an input function is assigned a distinct temporary in the function’s translation, and we replace the recursive type of programs P with a simpler type of pairs, where each pair consists of a list of flat functions and a flat main program expression. Every variable reference in the source program becomes either a temporary or a numeric index into the global list of functions.

Flattening works like closure conversion in instantiating PHOAS terms for use with de Bruijn levels. As this translation returns us to first-order languages, its correctness proof requires more lemmas about lists and maps, though most are independent of the set of language constructs. The main inductive theorems are comparable in complexity and proof organization to those for closure conversion.

4.5 Code Generation

The final compiler pass translates flat programs into assembly language. At this stage, neither source nor target language is novel. We still prove every theorem with an adaptive tactic program, but the basic organization of theorems is as one would expect from related projects. Our translation uses one register as the heap limit pointer, incrementing it as products, sums, and references are allocated. Another register doubles as the function argument register and as a place to stash short-lived values during double memory indirections. Finally, we reserve a register to store a recursive function’s “self” pointer. The remaining $N - 3$ registers are used to store the first $N - 3$ temporaries of each procedure. The remaining temporaries are stored in a global area at the beginning of memory. Since we deal only with compiling whole programs, it is easy to choose a size for this region by finding the highest numbered temporary used in a program.

The correctness proof is comparable in complexity to those of previous phases, but with significantly more supporting lemmas.

5. Optimizations

To better gauge how our approach scales to the algorithms used by real compilers, we also implemented and verified two common optimization passes.

5.1 Common Subexpression Elimination

Between closure conversion and flattening, we perform intraprocedural common subexpression elimination (CSE) on closed programs. Since our languages have no intraprocedural iteration constructs, there is no need to perform dataflow analysis. Instead, a single recursive traversal of a program suffices. The optimization still simplifies cases like application of a known function, where it is possible to avoid building a closure.

As we descend into a program’s structure, we maintain a mapping from variables to symbolic values, as defined below.

$$\text{Symbolic values } s ::= \#n \mid c \mid () \mid \langle s, s \rangle \mid \text{inl}(s) \mid \text{inr}(s)$$

Values not built from the basic constant, unit, product, and sum constructors are represented with symbolic variables $\#n$, where a fresh n is generated for each new input-program variable that cannot be determined to have more specific structure.

The purpose of CSE is to remove some redundant bindings and case analyses. This transformation may sound complicated enough to require conversion of input programs to first-order form to analyze them. However, it is possible to implement CSE in an elegant higher-order way. In translating a parametric program P , we must produce a CSE’d version of it for each possible variable representation `var`. Our solution is to do so by instantiating P at variable type `var * sval`, where `sval` is the type of symbolic values s .

Thus, each variable is tagged with a symbolic representation, and this representation may be accessed directly at use sites. The main translation maintains a mapping from symbolic values to variables. We use this mapping to simplify case expressions with discriminants that we see statically are either `inl` or `inr`. When proceeding under a let binder, the translation evaluates the bound expression symbolically. If the result is in the map, we avoid creating a new binder in the translation. Instead, we apply the binder body, which is a function over variable/value pairs, to the variable that our map associates with the appropriate symbolic value, paired with that value. If the value we are binding is not found in the map, we do create a new binder, and, in the recursive call inside the binder’s scope, we add the new variable to the symbolic map.

The main correctness theorem for this translation is proved very similarly to the main theorem for CPS conversion. The proof can be a bit simpler because we need no value compatibility relation; CSE has no effect on the values that appear during program evaluation. We prove the main theorem with about 20 lines of tactic code for performing appropriate case analyses, applying IHes and a lemma about primops, and materializing known facts about variables mentioned in expression equivalence derivations.

5.2 Combined Register Allocation and Dead Code Elimination

In a single pass performed between flattening and code generation, we combine register allocation and dead code elimination. Code generation automatically assigns the lowest-numbered temporaries to registers. Thus, the task of “register allocation” is simply to minimize the number of temporaries that each procedure uses, by finding opportunities to combine several mutually non-interfering temporaries into one. We use liveness information to calculate interference graphs. As with CSE, the lack of intraprocedural iteration makes it possible to compute an interference graph in a single traversal of procedure syntax. We use the same liveness information to eliminate useless assignments to temporaries.

Our implementation makes use of the finite set and map support in Coq’s standard library. We represent liveness information with sets of temporaries, interference graphs with sets of unordered pairs of temporaries, and temporary reassignments with maps from temporaries to temporaries. Coq’s library contains functors that build such structures from modules describing keys, and each functor output contains a set of standard theorems about its data structure. On top of this, we also implement and use an abstract data structure for temporary sets whose complements are finite, for use in choosing new names for temporaries.

As with code generation and many other kinds of low-level reasoning, this correctness proof is built from many unsurprising lemmas. By relying on the standard theorems about sets and maps, we

Component	Total	Proofs
Source language	228	0
Core PHOAS language	266	2
PHOASification	28	0
Correctness	390	138
Well-formedness	40	17
CPS language	279	18
CPS translation	94	0
Correctness	221	60
Well-formedness	39	12
Closed language	311	21
Closure conversion	303	13
Correctness	652	238
Well-formedness	261	119
CSE	87	1
Correctness	228	80
Well-formedness	177	70
Flat language	108	0
Flattening	63	0
Correctness	524	156
Register allocation	201	49
Correctness	642	310
Assembly language	105	0
Code generation	153	0
Correctness	1156	491
Overall compiler	13	0
Correctness	89	12
Total	6658	1807

Figure 10. Lines of code in different components

manage to avoid most proving that is not specific to our transformation.

6. Statistics

Figure 10 breaks down our development by the number of lines of code in each component. We include how many lines of code in each component come from proofs, which counts literal proof scripts, resolution hints, and definitions of tactic functions. More or less all of the remaining lines come from definitions of syntax and semantics, in the files corresponding to languages; from compiler phase implementations, in their files; or from theorem statements and auxiliary definitions, in “Correctness” and “Well-formedness” files.

Our development depends upon our Lambda Tamer Coq library, which contains about 1500 lines of object-language-agnostic theorems and tactics. We assert two axioms from the Coq standard library: functional extensionality, which says that two functions are equal if they map equal inputs to equal outputs; and proof irrelevance for equality proofs, which says that no equality proposition has more than one distinct proof. This pair of axioms has been proved on paper to be consistent with CIC.

The “Well-formedness” components in Figure 10 deal with proofs that transformations produce well-formed PHOAS terms from well-formed inputs. We conjecture that there are CIC-consistent axioms stating that all PHOAS terms are well-formed. If this were proved metatheoretically, then we could safely omit the well-formedness proofs.

Our first finished compiler was for our final source language minus let expressions, constants, equality testing, and recursive functions. We also proved a simpler version of the final correctness theorem, stating only that a compiled program exhibits the same

binary success-or-failure result as the original, ignoring details of returned values and thrown exceptions. As we added the enhancements needed to reach the final version, we measured how much effort was required.

6.1 Strengthening the Main Theorem

Since our source language is Turing-complete, the interesting aspects of compiler verification must be tackled even if the final theorem only distinguishes between two distinct classes of program outcome. Other distinctions may be modeled using tests within the object language. Thus, it was pragmatic for us to develop our initial proofs using this simplification. Later, we went back and adapted the proofs to allow us to prove Theorem 1 as we stated it earlier in the paper.

This required updating the different object languages so that halt and fail operations take parameters. We added or modified about 100 lines of syntax and semantics. We also had to introduce the “compiler data layout contract” relation and its relatives for the different translation phases, whose definitions added about 150 lines of unsurprising code. Additionally, we added about 100 lines of theorem statements, one-liner automated proofs, and resolution hints for some new theorems about the contract relations.

Beyond that, we modified or added about 80 lines of theorem statements and proof script, with most added to deal with new existential quantifications over program result values. Many of these changes were improvements that occurred to us as we worked on the upgrade, such that we would say that the changes belonged in the original compiler. Notably, the correctness proofs of the optimizations required no changes. We worked on this upgrade over the course of two days, with performance of the Coq proof assistant being the primary limiting factor. Automation of large proofs frequently leads Coq to run out of memory or run for excessively long, and we spent more time than we would like tuning our scripts to skirt these limitations on an old computer with modest resources. We believe that relatively straightforward improvements to Coq’s proof engine would make this kind of upgrade quite reasonable to complete in well under a day of work. Even with the current state of the tools, the upgrade did not require us to add any proof code specific to a particular case of any of our inductive proofs.

6.2 Adding let Expressions

Adding let expressions was relatively simple, since general let only appears until CPS translation, and since let does not add a new category of runtime values. Thus, we extended the first two translations only and the syntax and semantics of the first two languages. We also added a let case to the expression compatibility relation in the PHOASification correctness proof. These changes amounted to about 30 new lines of code, with no old lines modified. All of our proofs continued working unchanged. We did not need to update a single theorem statement or proof script. The whole process took under half an hour.

6.3 Adding Constants and Equality Testing

Next, we added constants and equality comparison of constants, which impacts much more of the compiler and its proof. We added or changed about 100 lines defining syntax, semantics, and translations, and we added about 50 lines defining new rules for inductive relations used in the proofs. To adapt the old proofs, we had to change some patterns to mention new constructors of datatypes, to placate Coq’s limited support for inferring which datatype is being pattern-matched on. We proved one lemma about the encoding of constants, giving it a one-line proof and adding it as a hint, and we added one additional one-line hint that detects when constants are being compared for equality and then performs case analysis on whether they are equal. These proof changes amount to about

10 lines in total, and we also added 5 more lines to improve performance in a way not related to constants. We spent about half a day on this extension, again with most of that time spent waiting for slow proof search to finish.

6.4 Adding Recursive Functions

Replacing non-recursive anonymous functions $\lambda x. e$ with recursive anonymous functions $\text{fix } f(x). e$ turned out to be the most intensive change. We added or modified about 50 lines to account for additions to syntax, semantics, and translations, and we modified about 20 lines that define function abstraction rules for further inductive relations.

We added or modified about 350 lines of theorem statements and proofs. In each of the earlier phases of the compiler, we modified at most one line of proof in a way that is really specific to recursive functions. The remaining extra lines come from the fact that fix is our only construct that binds more than one variable at once. We had already proved a host of lemmas specialized to the case of one variable at a time, and we needed to duplicate these lemmas, reusing their automated proofs without changes. Other alterations to theorem statements and tactics reflected only the need to handle a new binding pattern. The exception to this trend was code generation, where a change to the calling convention triggered a fair amount of churn in theorem statements. The full upgrade to fix support took us approximately one day.

7. Related Work

Compiler verification for first-order languages has a considerable history, but we will focus on reviewing related work in compiling functional languages. See the bibliography by Dave (2003) for pointers into the traditional literature on first-order compilers. There have been a variety of investigations into verifying compilers for pure functional languages, drawing on several very different representation and proof techniques.

Flatau (1992) described a partial verification of a compiler from a subset of the Nqthm logic to the first-order imperative language Piton, performed with the Nqthm prover. The proof was highly automated, in the usual style of Nqthm and ACL2. Since the compiler worked in a single pass and targeted a first-order language, it avoided most of the issues inherent in reasoning about rearranging variable binders.

Minamide and Okuma (2003) used Isabelle/HOL and Isar to build proofs of correctness for CPS translations for bare-bones untyped lambda calculus, using concrete encoding of binders with variable names. The simplest translation that they verified (that of Plotkin) had a correctness proof of about 250 lines. Their proof for Danvy and Nielsen’s translation took about 400 lines, and this figure grew to about 600 when they added let binding to the object language.

Tian (2006) used Twelf with HOAS to prove CPS translation correctness for an untyped, pure mini-ML language without recursion, which is subsumed within our case study source language. His comparable CPS translation theorem took about 50 lines, in the usual, fully manual Twelf style. Tian’s development uses a target language more specialized to his particular translation, featuring hardcoded binding of a second-class continuation with any function definition, rather than building this functionality on top of product types and first-class continuations. We expect that the cost of writing such proofs in Twelf becomes more apparent when source and target languages are less tightly coupled, so that some inductive proof cases must build proof trees of non-trivial depth.

In past work (Chlipala 2007), we verified a compiler from basic simply-typed lambda calculus to a target language similar to three-address code. We used the dependent de Bruijn representation throughout the early stages of the compiler, and we relied on a

metaprogramming component to prove some of the standard binder lemmas for us. By switching to PHOAS, we have avoided the need to state those lemmas explicitly. Our present compiler extends our past PHOAS-based work (Chlipala 2008) by treating impurity, recursion, and the rest of the compilation pipeline beyond CPS translation and closure conversion.

Dargaye and Leroy (2007) used Coq to verify CPS translations for another mini-ML language encoded with de Bruijn indices. We believe that our CPS translation is comparable in functionality to their optimized translation, with the exception that ours does not do tail call optimization. They give code size statistics for their CPS translation, broken into “specifications” and “proofs.” The “proofs” size for the dependencies of the optimized transform correctness proof is 4287 lines, an order of magnitude more than the total size of our CPS correctness file. The complexities of the languages treated by the two projects are not directly comparable; Dargaye and Leroy include variable-arity recursive functions and datatype constructors, but they omit impure features.

Benton and Hur (2009) take a very different approach to compiler verification in Coq, starting with a typed functional language (represented with de Bruijn indices) and using step-indexed logical relations over types to establish correctness. Their compiler works in one pass, so the theorems involved are very different from those in the early phases of our compiler. Their development uses the usual Coq manual proof style and runs to about 4000 lines. Though their source language (basic lambda calculus with recursive functions) is less featureful than the source language of this paper, their final correctness theorem is more general than in any related work, as it facilitates sound linking with code produced by different compilers or written by hand.

8. Conclusion

Mostly-manual interactive proving of theorems about programming languages and compilers can be a very engaging challenge, and it has been a crucial tool in our efforts to figure out the right abstractions for the job. Nonetheless, we do not believe that this style scales to real-world implementations of high-level languages. We think it is not at all too early to be thinking about the techniques that may some day be applied in such a setting. Our case study in this paper is a verified compiler whose mechanized correctness proofs are tactic programs that can adapt automatically to specification changes. We believe strongly that, if compiler verification ever goes mainstream, it will be done more like we propose here than like the styles more commonly employed by languages researchers.

To avoid getting bogged down in administrative lemmas about binding, we exploit the parametric higher-order abstract syntax encoding, along with a new way of using it to encode substitution-free operational semantics. As a result, some of our compiler phase correctness proofs are shorter even than what a diligent semanticist would write on paper. Despite our use of novel representations, our final theorem is stated only in terms of established encodings and relies on no nonstandard axioms.

We hope to expand our case study into a verified compiler for core Standard ML. This requires adding a few more features to the source language and implementing a (hopefully straightforward) elaboration from the official abstract syntax of Standard ML. We would also like to define typing judgments for each language and prove a type preservation theorem for the final compiler, which could output Typed Assembly Language. We conjecture that this extension would require little change to the semantic preservation theorems and proofs. It would also be interesting to try to complete the end-to-end compiler picture with a verified parser, type inference engine, and assembler.

Acknowledgments

We would like to thank Greg Morrisett, Ryan Wisnesky, and the anonymous referees for helpful feedback on earlier versions of this paper. This work was supported by a gift from Microsoft Research.

References

- Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proc. POPL*, pages 3–15, 2008.
- Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. ICFP*, pages 97–108, 2009.
- Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. PLDI*, pages 54–65, 2007.
- Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proc. ICFP*, pages 143–156, 2008.
- Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
- Zaynab Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Proc. LPAR*, pages 211–225, 2007.
- Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.
- Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
- David Delahaye. A tactic language for the system Coq. In *Proc. LPAR*, pages 85–95, 2000.
- Arthur D. Flatau. *A Verified Implementation of an Applicative Language with Dynamic Storage Allocation*. PhD thesis, University of Texas at Austin, November 1992.
- Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In *Proc. ICFP*, pages 75–86, 2008.
- Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. of the ACM*, 40(1):143–184, 1993.
- Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *Proc. ICALP*, pages 963–978, 2001.
- Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54, 2006.
- Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.
- Yasuhiko Minamide and Koji Okuma. Verifying CPS transformations in Isabelle/HOL. In *Proc. MERLIN*, pages 1–8, 2003.
- J Strother Moore. A mechanically verified language implementation. *J. Automated Reasoning*, 5(4):461–492, 1989.
- G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. FPCA*, pages 66–77, 1995.
- F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. PLDI*, pages 199–208, 1988.
- Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. CADE*, pages 202–206, 1999.
- Ye Henry Tian. Mechanically verifying correctness of CPS compilation. In *Proc. CATS*, pages 41–51, 2006.
- C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In *Proc. CADE*, pages 38–53, 2005.
- Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008.
- Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proc. TPHOLs*, pages 167–184, 1999.

Data groups: Specifying the modification of extended state

K. Rustan M. Leino

Compaq Systems Research Center

130 Lytton Ave., Palo Alto, CA 94301, U.S.A.

www.research.digital.com/SRC/people/Rustan_Leino

rustan@pa.dec.com

Abstract

This paper explores the interpretation of specifications in the context of an object-oriented programming language with subclassing and method overrides. In particular, the paper considers annotations for describing what variables a method may change and the interpretation of these annotations. The paper shows that there is a problem to be solved in the specification of methods whose overrides may modify additional state introduced in subclasses. As a solution to this problem, the paper introduces *data groups*, which enable modular checking and rather naturally capture a programmer's design decisions.

0 Introduction

Specifications help in the documentation of computer programs. Ideally, specifications can be used by a mechanical program analyzer to check the body of a method against its specification, attempting to find errors. The Extended Static Checkers for Modula-3 [DLNS98, LN98b, Det96] and for Java [ESC], which work on object-oriented programs, are examples of such program checkers.

This paper concerns the specification of methods. A method specification is a contract between the implementation of a method and its callers. As such, it includes a *precondition*, which documents what a caller must establish before invoking the method. Consequently, the implementation can assume the precondition on entry to the method body. A method specification also includes a *postcondition*, which documents what the implementation must establish on exit. Consequently, the caller can assume the postcondition upon return from the method invocation. When reasoning about method implementations and calls, only the contract given by the specification is used. That is, one does

not use the code in a method's callers when reasoning about the method implementation, and one does not use the implementation when reasoning about the calls.

To be useful to the caller, it is important that the postcondition of a method detail what variables the method does not change. But since the scope of the caller can include variables that are not visible in the scope where the method is declared and specified, it is not possible to explicitly list all unchanged variables in the method's postcondition. Instead, the annotation language must include some form of syntactic shorthand ("sugar") whose interpretation as part of the postcondition is a function of the scope in which it is interpreted. A nice construct for this is the **modifies** clause, which lists those variables that the method is allowed to modify, thereby specifying that the method does not modify any other variables [GH93]. For example, suppose that the specification of a method *m* occurs in a scope where two variables, *x* and *y*, are visible, and that the specification includes the **modifies** clause

modifies x

If *m* is called from a scope where, additionally, a variable *z* is visible, then the caller's interpretation ("desugaring") of the specification says that the call may possibly modify *x*, but leaves both *y* and *z* unchanged.

The fact that a **modifies** clause is interpreted differently in different scopes raises a concern about *modular soundness* [Lei95]. For the purpose of this paper, modular soundness means that the implementation, which is checked to meet the specification as interpreted in the scope containing the method body, actually lives up to a caller's expectations, which are based on the specification as interpreted in the scope of the call. A consequence of modular soundness is that one can check a class even in the absence of its future clients and subclasses.

This paper explores the interpretation of specifications in the context of an object-oriented programming language with subclassing and method overrides, for example like Java. In particular, I consider annotations for describing what a method may change and the interpretation of these annotations. I show that there is a problem to be solved in the

specification of methods whose overrides may modify additional state introduced in subclasses. As a solution to this problem, I introduce *data groups*, which adhere to modular soundness and rather naturally capture a programmer's design decisions.

For simplicity, I restrict my attention to the operations on only one object, the implicit *self* parameter. Nevertheless, because of inheritance and method overriding, the implementations of the methods of this object may be found in superclasses and subclasses of the class being checked.

1 Extending the state of a superclass

To illustrate the problem, I introduce a simplified example of a computer arcade game—an excellent application of object-oriented programming indeed.

The design centers around *sprites*. A sprite is a game object that appears somewhere on the screen. In this simple example, every sprite has a position, a color, and methods to update these. The main program, which I will not show, essentially consists of a loop that performs one iteration per video frame. Each iteration works in two phases. The first phase invokes the `update` method on each sprite, which updates the sprite's position, color, and other attributes. The second phase invokes the `draw` method on each sprite, which renders the sprite on the screen.

Here is the declaration of class `Sprite`, in which the methods have been annotated with **modifies** clauses:

```
class Sprite {
    int x, y;
    void updatePosition() /* modifies x, y */
    {
    }
    int col;
    void updateColor() /* modifies col */
    {
    }
    void update() /* modifies x, y, col */
    {
        updatePosition(); updateColor();
    }
    void draw() /* modifies (nothing) */
    {
    }
}
```

The default `update` method invokes the `updatePosition` and `updateColor` methods, whose default implementations do nothing. Any of these methods can be overridden in `Sprite` subclasses. For example, a moving sprite that never changes colors would override the `updatePosition` method, a stationary sprite whose color changes over time would override the `updateColor` method, and a sprite that adds further attributes that need to be updated overrides the `update` method and possibly also the `updatePosition` and `updateColor` methods.

Since the specifications I have given in the example show only **modifies** clauses, checking that an implementation

meets its specification comes down to checking that it modifies only those variables that it is permitted to modify. The implementations of the `updatePosition`, `updateColor`, and `draw` methods are no-ops, so they trivially satisfy their specifications. The `update` method invokes the other two `update` methods, whose **modifies** clauses say they may modify `x`, `y`, and `col`. So `update` in effect modifies `x`, `y`, and `col`, and this is exactly what its specification allows. We conclude that the methods in class `Sprite` meet their specifications.

Let us now consider a subclass `Hero` of `Sprite`, representing the hero of the game. The hero can move about, and hence the `Hero` class provides its own implementation of the `updatePosition` method by overriding this method. The next position of the hero is calculated from the hero's velocity and acceleration, which are represented as instance variables. The `Hero` class is declared as follows:

```
class Hero extends Sprite {
    int dx, dy;
    int ddx, ddy;
    void updatePosition()
    {
        x += dx + ddx/2; y += dy + ddy/2;
        dx += ddx; dy += ddy;
    }
    ...
}
```

The `Hero` implementation of `updatePosition` increases `x` and `y` by appropriate amounts ($\Delta d = v_0 \cdot t + \frac{1}{2} \cdot a \cdot t^2$ where $t = 1$). In addition, it updates the velocity according to the current acceleration. (Omitted from this example is the update of acceleration, which is computed according to the game player's joystick movements.) It seems natural to update the velocity in the method that calculates the new position, but the specification of `updatePosition` (given in class `Sprite`) allows only `x` and `y` to be modified, not `dx` and `dy` which are not even defined in class `Sprite`. (If the update of `dx` and `dy` instead took place in method `update`, there would still be a problem, since the **modifies** clause of `update` also does not include these variables.)

As evidenced in this example, the reason for overriding a method is not just to change what the method does algorithmically, but also to change what data the method updates. In fact, the main reason for designing a subclass is to introduce subclass-specific variables, and it is the uses and updates of such variables that necessitate being able to override methods. For example, class `Sprite` was designed with the intention that subclasses be able to add sprite attributes and update these in appropriate methods. So how does one in a superclass write the specification of a method such that subclasses can extend the superclass's state (that is, introduce additional variables) and override the method to modify this extended state?

2 Three straw man proposals

In this section, I discuss three proposals that I often hear suggested for solving the problem of specifying the modification of extended state. I show that these proposals don't work. This is what it means for a proposal to work:

- the proposal must provide a way to annotate classes like `Sprite` and `Hero` such that the desired method implementations in these classes will meet their specifications,
- the interpretation of specifications must be useful to callers (for example, specifications should not all be treated as "can do anything whatsoever"),
- the annotations should not be unnecessarily tedious to write down, and
- the proposal must adhere to modular soundness.

Here is the first proposal:

Straw man 0. A subclass can *refine* the specification of a method when it overrides it. That is, a subclass can *weaken* the precondition of the method in the superclass (that is, say that the overridden method implementation will work in more situations) and *strengthen* the postcondition (that is, be more specific about the effect of the method).

It is well known that this proposal is sound. However, it doesn't solve the problem at hand. To strengthen the postcondition means to be more precise about the final values of variables. This is just the opposite of what we'd like—we'd like the new postcondition to allow more variables to be modified, that is, to put no restrictions at all on the final values of these variables. Stated differently, while *shrinking* the list in the **modifies** clause is sound, *enlarging* it is what we want when specifying a subclass's method overrides.

Another straw man proposal is the following:

Straw man 1. Let `m` be a method declared and specified in a class `T`. An implementation of `m` is allowed to modify those variables listed in the **modifies** clause of `m`, plus any variable declared in any proper subtype of `T`.

Although sound, this straw man is too liberal about the modification of variables in subclasses. In fact, a subclass loses the advantage of **modifies** clauses with this proposal. To illustrate, I will show an example that builds on class `Sprite`.

Consider a class of *monsters* with a strength attribute. Rather than storing this attribute as an instance variable in every monster object, suppose a class `Monster` has a method that returns the value of the strength attribute. Thus, different `Monster` subclasses can decide on their own representation of the strength attribute. For example, if the strength of a class of monsters is constant, the method can return that

constant, without taking up any per-object storage. This design trades quick access of an attribute for flexibility in how the attribute is represented.

The following declaration shows class `Monster`, which uses the strength attribute in updating the sprite position.

```
class Monster extends Sprite {  
    int getStrength() /* modifies (nothing) */  
    { return 100; }  
    void updatePosition()  
    { if (getStrength() < 10) {  
        x += 2;  
    } else {  
        x += 4;  
    } }  
}
```

A particular `Monster` subclass is `AgingMonster`, which adds an age attribute and overrides the `draw` method so as to render the monster differently according to its strength-to-age ratio.

```
class AgingMonster extends Monster {  
    int age;  
    ...  
    void draw()  
    { int bitmapID;  
        if (age == 0) {  
            bitmapID = MONSTER_INFANT;  
        } else {  
            int s = getStrength();  
            int relativeStrength = s/age;  
            if (relativeStrength < 5) {  
                bitmapID = MONSTER_WIMPY;  
            } elseif (relativeStrength < 10) {  
                bitmapID = MONSTER_NORMAL;  
            } else {  
                bitmapID = MONSTER_STRONG;  
            } }  
        Bitmap.Draw(x, y, bitmapID);  
    }  
}
```

The name `Bitmap.Draw` denotes some procedure that can draw a bitmap given a screen coordinate and an ID.

The correctness of the `AgingMonster` implementation of `draw` relies on the fact that the call to `getStrength` does not modify `age`. In particular, if `getStrength` were to set `age` to 0, then the computation of `relativeStrength` would result in a division-by-zero error. The `getStrength` method is specified with an empty **modifies** clause, but Straw Man 1 gives implementations of `getStrength` permission to modify `age`, since `age` is declared in a proper subclass of `Monster`. Thus, the interpreted specification for method `getStrength` is not strong enough for one to conclude that method `draw` will execute correctly.

There is a workaround. If a class is allowed to refine the specifications of methods declared in superclasses, class `AgingMonster` can strengthen the postcondition of method `getStrength` with `agepre == agepost`. But this would quickly get annoying, because programmers would then sometimes rely on the absence of `age` in the **modifies** clause to conclude that `age` is not changed, and sometimes rely on an explicit postcondition `agepre == agepost` to conclude the same thing. Even worse, strengthening the specification of all methods declared in a superclass whenever a class introduces new variables would quickly grow to be an unacceptably tedious chore.

The next straw man proposal seeks to alleviate this chore by making the mentioned postcondition strengthening the default interpretation, and providing a new specification construct **also-modifies** that can override the default interpretation:

Straw man 2. Let `m` be a method declared and specified in a class `T`. An implementation of `m` in a subclass `U` of `T` is allowed to modify those variables listed in the **modifies** clause of `m` as given in class `T`, plus any variable declared in any **also-modifies** clause for `m` as given in some superclass of `U`.

This straw man seems to solve the problem for the `Hero` example: One would simply annotate the `updatePosition` override with

also-modifies `dx, dy`

This would give the `updatePosition` implementation in `Hero` permission to modify not just `x` and `y` (as granted by the original specification of `updatePosition` in `Sprite`), but also the variables `dx` and `dy`. (One could also add `ddx` and `ddy` to the **also-modifies** clause, if desired.)

Let us consider how Straw Man 2 stands up to modular soundness. Suppose that the game uses one hero object throughout many game levels. As a new level starts, the program will call a method `startNewLevel` on the hero object. This method resets certain attributes of the hero object while leaving other attributes unchanged, preparing it to begin the new level. To this end, suppose class `Hero` contains the following method declaration and specification, where the keyword **ensures** is used to express a given postcondition:

```
void startNewLevel()
/* modifies x, y, col, dx, dy, ddx, ddy
   ensures dxpost == 0 ∧ dypost == 0 */
{ dx = 0; dy = 0;
  update();
}
```

The given implementation of `startNewLevel` contains an error: The invocation of `update` results in a call to the `update` implementation in class `Sprite`, whose invocation

of `updatePosition` in turn results in a call to the implementation of `updatePosition` given in class `Hero` (because of dynamic method dispatch). This implementation of `updatePosition` modifies the `dx` and `dy` variables. Thus, executions of `startNewLevel` may well end with non-zero values for `dx` and `dy`, so the implementation of method `startNewLevel` does not meet its specification.

Unfortunately, the methodology proposed by Straw Man 2 does not allow one to catch the error in `startNewLevel`. The problem is that even though the interpretation of the specification of `updatePosition` in class `Hero` reveals that `dx` and `dy` may be modified (since the **also-modifies** annotation of `updatePosition` in class `Hero` lists these variables), the `update` method is not overridden in `Hero` and thus gets its specification solely from the one given in class `Sprite`. Hence, the interpretation of the specification of `update` shows `dx` and `dy` as being unchanged, so a program checker will not find anything wrong with the implementation of `startNewLevel`.

Note that the implementations in class `Sprite` do meet their specifications under Straw Man 2. For example, the interpretation of the specification of `updatePosition` in class `Sprite` includes only `x` and `y`, both of which are allowed to be modified also by the implementation of `update`. Hence, there is no error for the checker to report in class `Sprite` either.

In conclusion, Straw Man 2 seems pretty good at first, but since it allows the specifications of different methods (in the example, `updatePosition` and `update`) to be extended in different ways (by having different **also-modifies** clauses, or none at all), the proposal does not adhere to modular soundness. The proposal in the next section provides annotations for data rather than for methods, the effect of which is to make specification extensions apply in a uniform manner.

3 Data groups

In this section, I explain my proposal and demonstrate how it solves the problems with the examples shown previously. In Section 4, I show how a program checker can enforce the proposal, and in Section 5, I argue that my proposal is sound.

The idea is to introduce *data groups*, which represent sets of variables. A data group is declared in a class, just like an instance variable is. The declaration of an instance variable is annotated with the names of the data groups to which the variable belongs. Data groups can be nested, that is, a group can be declared as a member of another group. A data group can be listed in a **modifies** clause, where it represents the set of all members of the group.

Using data groups, the declaration of `Sprite` can be

written as:

```
class Sprite {
    /* group attributes; */
    /* group position member-of attributes; */
    int x /* member-of position */;
    int y /* member-of position */;
    void updatePosition() /* modifies position */
    {
    }
    /* group color member-of attributes; */
    int col /* member-of color */;
    void updateColor() /* modifies color */
    {
    }
    void update() /* modifies attributes */
    {
        updatePosition(); updateColor();
    }
    /* group drawState; */
    void draw() /* modifies drawState */
    {
    }
}
```

This version of class `Sprite` declares four data groups, `attributes`, `position`, `color`, and `drawState`, and declares `position` and `color` to be members of `attributes`, `x` and `y` to be members of `position`, and `col` to be a member of `color`. Class `Sprite` does not declare any members of group `drawState`.

Since `updatePosition` is declared with the specification `modifies position`, an implementation of this method is allowed to modify `x` and `y`. In addition, an implementation of this method is allowed to modify any variables declared in `Sprite` subclasses to be members of `position`. An implementation of `updatePosition` is not allowed to call method `updateColor`, for example, since `color` is not a member of `position`.

By introducing a data group `drawState` and listing it in the `modifies` clause of method `draw`, implementations of `draw` in `Sprite` subclasses are given a way to modify instance variables (in particular, to modify variables that are introduced as members of `drawState`).

The following illustrates how one can use data groups to annotate class `Hero`:

```
class Hero extends Sprite {
    int dx /* member-of position */;
    int dy /* member-of position */;
    int ddx /* member-of position */;
    int ddy /* member-of position */;
    void updatePosition()
    {
        x += dx + ddx/2; y += dy + ddy/2;
        dx += ddx; dy += ddy;
    }
}
```

```
void startNewLevel()
/* modifies attributes
ensures dxpost == 0 ∧ dypost == 0 */
{ dx = 0; dy = 0;
    update();
}
```

The override of `updatePosition` gets its permission to modify `dx` and `dy` from the fact that these variables are members of the data group `position`. This solves the problem of how to specify `updatePosition` in class `Sprite` so that a subclass like `Hero` can modify the state it introduces.

With data groups, the error in `startNewLevel` is detected. Since `dx` and `dy` are members of `position`, which in turn is a member of `attributes`, a program checker will know that `dx` and `dy` may be modified as a result of invoking `update`. Since the specification of `update` says nothing further about the final values of `dx` and `dy`, one cannot conclude that they remain 0 after the call.

As for the `AgingMonster` example, the data groups proposal does allow one to infer that no division-by-zero error is incurred in the evaluation of `s/age`: The guarding `if else` statement guarantees that `age` is non-zero before the call to `getStrength`, and since `age` is not modified by `getStrength`, whose `modifies` clause is empty, `age` remains non-zero on return from `getStrength`.

I will give two more examples that illustrate the use of data groups.

First, note that the members of two groups are allowed to overlap, that is, that a variable is allowed to be a member of several groups. For example, if a `Sprite` subclass declares a variable

```
int k /* member-of position, drawState */;
```

then `k` can be modified by any of the methods `update`, `updatePosition`, and `draw`.

Second, I give another example to illustrate that it is useful to allow groups to contain other groups. Suppose a subclass of `Sprite`, `Centipede`, introduces a `legs` attribute. Class `Centipede` declares a data group `legs` and a method `updateLegs` with license to modify `legs`, which implies the license to modify the members of `legs`. By declaring `legs` as a member of `attributes`, the `update` method gets permission to call method `updateLegs`:

```
class Centipede extends Sprite {
    /* group legs member-of attributes; */
    int legCount /* member-of legs */;
    void updateLegs() /* modifies legs */
    {
        legCount = ...;
    }
    void update()
    {
        updatePosition(); updateColor();
        updateLegs();
    }
}
```

4 Enforcing the data groups proposal

This section describes more precisely how a program checker handles data groups.

For every data group g , the checker introduces a new variable $g\text{Residue}$. This so-called *residue variable* is used to represent those of g 's members that are not in scope—in a modular program, there is always a possibility of a future subclass introducing a new variable as a member of a previously declared group.

To interpret a **modifies** clause

modifies w

the checker first replaces w with the variables in the *downward closure* of w . For any set of variables and data groups w , the downward closure of w , written $\text{down}(w)$, is defined as the smallest superset of w such that for any group g in $\text{down}(w)$, $g\text{Residue}$ and the variables and groups declared with

member-of g

are also in $\text{down}(w)$.

For example, computing the downward closure of the modifies list attributes in class `Hero` as shown in Section 3 yields

```
attributes, attributesResidue,  
position, positionResidue, x, y, dx, dy, ddx, ddy,  
color, colorResidue, col
```

Thus, in that class,

modifies attributes

is interpreted as

```
modifies attributesResidue, positionResidue,  
x, y, dx, dy, ddx, ddy,  
colorResidue, col
```

By handling data groups in the way described, the `Hero` implementation of method `startNewLevel`, for example, is allowed to modify `dx` and `dy` and is allowed to call method `update` (but the assignments to `dx` and `dy` must take place *after* the call to `update` in order to establish the specified postcondition of `startNewLevel`). The implementation of `startNewLevel` would also be allowed to call, for example, `updatePosition` directly. But the checker would complain if `startNewLevel` called `draw`, because the call to `draw` would be treated as modifying the residue variable `drawStateResidue`, and that variable is not in the downward closure of `attributes`.

5 Soundness

The key to making the data groups proposal sound is that it is always known to which groups a given variable or group belongs, and that residue variables are used to represent members of the group that are not in scope. The data groups proposal is, in fact, a variation of the use of abstract variables and dependencies in my thesis [Lei95]. I will explain the relation between the two approaches in this section, and relegate the proof of soundness to that for dependencies in my thesis.

A data group is like an *abstract variable*. An abstract variable (also called a *specification variable*) is a fictitious variable introduced for the purpose of writing specifications. The value of an abstract variable is represented in terms of program variables and other abstract variables. In some scopes, it is not possible, nor desirable, to specify the representation of an abstract variable because not all of the variables of the representation are visible. This tends to happen often in object-oriented programs, where the representation is often subclass-specific. However, if the abstract variable and *some* of the variables of the representation are visible in a scope, then the fact that there is a dependency between these variables must be known to a program checker in order to achieve modular soundness. Consequently, an annotation language that admits abstract variables must also include some construct by which one can explicitly declare the dependency of an abstract variable on a variable that is part of its representation. For example, if `position` were an abstract variable, then

depends position **on** x

would declare that variable x is part of the representation of `position`. My thesis introduced such dependency declarations. The corresponding notion in this paper is the annotation that declares that x is a member of the data group `position`:

```
int x /* member-of position */;
```

Using dependencies, one can give a precise definition of what the occurrence of an abstract variable in a **modifies** clause means. For dependencies like the ones shown here, this interpretation is the same as that defined for data groups above: the downward closure.

My thesis contains a proof that the use of dependencies in this way adheres to modular soundness, provided the program meets two requirements and provided the interpretation includes residue variables. The two requirements, called the *visibility and authenticity requirements*, together state essentially that a dependency declaration

depends a **on** c

should be placed near the declaration of c , that is, so that every scope that includes the declaration of c also includes

the dependency declaration. Because the **member-of** annotation is made part of the declaration of the variable whose group membership it declares, the two requirements are automatically satisfied.

There is one other difference between data groups and abstract variables with dependencies. Suppose an abstract variable **a** depends on a variable **c**, and that the downward closure of the **modifies** clause of a method includes **c** but not **a**. The interpretation of such a **modifies** clause says that **c** may be modified, but only in such ways as to not change the abstract value of **a** [Lei95]. This is called a *side effect constraint* on **a**.

But with data groups, it would be meaningless to use side effect constraints, since data groups don't have values. Thus, if variable **c** is a member of a data group **a** and the downward closure of a method **m** includes **c** but not **a**, then the **modifies** clause does not constrain the implementation of **m** in how **c** is changed. Violations of modular soundness result from the deficiency that the different interpretations of a specification in different scopes are inconsistent. So by removing side effect constraints in *all* scopes, modular soundness is preserved.

From our experience with writing specifications for extended static checking, we have found it useful to introduce an abstract variable conventionally called **state** [LN98a]. This variable is declared to depend on variables representing the state of a class or module. The **state** variable is used in many **modifies** clauses, but not in pre- and postconditions. Furthermore, **state** is never given an exact definition in terms of its dependencies. Thus, the type of **state** is never important, so we declared its type to be **any**, where **any** is a new keyword that we added to the annotation language.

The data groups proposal grew from a feeling that it was a mistake to apply the side effect constraint on variables like **state** whose type is **any**—after all, the exact value of such a variable is never defined and thus cannot be relied on by any part of the program. By changing the checking methodology to not apply side effect constraints on variables of type **any**, one arrives at the interpretation of data groups presented in this paper.

As a final note on modular soundness, I mention without going into details that the absence of side effect constraints makes the authenticity requirement unnecessary. This means that it would be sound to declare the members of a data group at the time the group is declared, rather than declaring, at the time a variable is declared, of which groups the variable is a member. For example, instead of writing

```
/* group g; */
...
int x /* member-of g */;
```

one could write

```
int x;
...
/* group g contains x, ... */
```

Using **contains** in this way adheres to modular soundness (but declaring a group with both a **contains** and a **member-of** phrase does not). However, while introducing a group containing previously declared variables is sound and may occasionally be convenient, it does not solve the problem described in this paper.

6 Concluding remarks

In summary, this paper has introduced *data groups* as a natural way to document object-oriented programs. Data groups represent sets of variables and can be listed in the **modifies** clauses that document what methods are allowed to modify. The license to modify a data group implies the license to modify the members of the data group as defined by the *downward closure* rule.

Since data groups are closely related to the use of abstract variables and dependencies [Lei95], they adhere to the useful property of *modular soundness*, which implies that one can check a program one class at a time, without needing global program information. Although the literature has dealt extensively with data abstraction and refinement, including Hoare's famous 1972 paper [Hoa72], it seems that only my thesis and my work with Nelson [LN98a] have addressed the problem of having abstract variables in **modifies** clauses in a way that modern object-oriented programs tend to use them.

The use of data groups shown in this paper corresponds to *static*, as opposed to *dynamic*, dependencies. Dynamic dependencies arise when one class is implemented in terms of another. Achieving soundness with dynamic dependencies is more difficult than the case for static dependencies [LN98a, DLN98].

Data groups can be combined with abstract variables and dependencies. This is useful if one is interested in the abstract values of some attributes and in the representation functions defining these abstract values.

A related methodological approach to structuring the instance variables and methods of a class is *method groups*, first described by Lamping [Lam93] and developed further by Stata [Sta97]. Method groups and data groups both provide ways to organize and think about the variables declared in classes. Other than that, methods groups and data groups have different aims. The aim of method groups is to allow the variables declared in a superclass to be used in a different way in a subclass, a feature achieved by the following discipline: The variables and methods of a class are partitioned into method groups. A variable **x** in a method group **A** is allowed to be modified directly only by the methods in group

A ; methods in other groups can modify x only via calls to methods in group A . If a designer of a subclass chooses to replace a variable or method of a method group, all variables and methods of the method group must be replaced. The use of method groups can complement the use of data groups, whose aim is to address not *how* variables are used but rather the more fundamental question of *which* variables are allowed to be changed by which methods. If one wants to write specifications in terms of abstract values and allow subclasses to change the representation functions of these abstract values, then one can combine data groups, abstract variables, and dependencies with method groups.

A related approach to specifying in a superclass what a subclass method override is allowed to modify is using *region promises* [CBS98]. These are used in reasoning about software transformations. In contrast to data groups, the sets of variables included in different regions are required to be disjoint. This restriction facilitates reasoning about when two method calls can be commuted, but burdens the programmer with having to invent a partition on the class variables, which isn't always as natural.

The region promises are used in both **modifies** clauses and so-called **reads** clauses, which specify which variables a method is allowed to read. Although not explored in this paper, it seems that data groups may be as useful in **reads** clauses as they are in **modifies** clauses.

A complementary technique for finding errors in programs is explored by Jackson in his Aspect system [Jac95]. To give a crude comparison, Aspect features annotations with which one specifies what a method *must* modify, whereas the **modifies** clauses considered in this paper specify what a method is *allowed* to modify. To specify what a method must modify, one uses *aspects*, which are abstract entities that can be declared to have *dependences*, consisting of variables and other dependences. Such aspects are analogous to data groups.

There are many specification languages for documenting object-oriented software, including Larch/C++ [Lea96] and the specification languages surveyed by Lano and Haughton [LH94]. These specification languages do not, however, establish a formal connection between specifications and actual code. Without such a connection, one cannot build a programming tool for finding errors in implementations. As soon as one becomes interested in checking a method implementation against a specification that is useful to callers, one becomes concerned with what the implementation is allowed to modify. Add subclassing to the stew and one faces the problem described in this paper.

To motivate data groups in this paper, I spoke informally about the semantics of the example code. There are several Hoare-like logics and axiomatic semantics of object-oriented programs that define the semantics formally [Lea89, AdB94, Nau94, AL97, Lei97, PHM98, Lei98a]. Four of these [AL97, Lei97, PHM98, Lei98a] deal with programs

where objects are references to mutable data fields (instance variables) and method invocations are dynamically dispatched. However, except for Ecstatic [Lei97], these logics have focused more on the axiomatization of language features and object types than on the desugaring of useful specification constructs.

In the grand scheme of annotating object-oriented programs in ways that not only help programmers, but that also can be used by program analyzers, this paper has touched only on the modification of extended state. Though they sometimes seem like a nuisance in the specification of programs, **modifies** clauses are what give a checker precision across procedure boundaries. Vandevenoerde has also found **modifies** clauses to be useful in improving program performance [Van94].

Other important method annotations include pre- and post-conditions, of which useful variations have also been studied [Jon91, LB97]. As for annotating data, object invariants [Mey88, LW94, LH94, Lea96] is a concept useful to programmers and amenable as annotations accepted by a program checker. Like the modification of extended state, achieving modular soundness with object invariants is an issue [LS97].

Acknowledgements

Raymie Stata, Greg Nelson, Mark Lillibridge, and Martín Abadi made useful comments on a draft of this paper. The paper has also benefited from the workshop on Foundations of Object-Oriented Languages [Lei98b].

References

- [AdB94] Pierre America and Frank de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.
- [AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidot and Max Dauchet, editors, *Theory and Practice of Software Development: Proceedings / TAP-SOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer, April 1997.
- [CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE'98)*, pages 167–176. IEEE Computer Society, April 1998.

- [Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM SIGSOFT, January 1996.
- [DLN98] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Compaq Systems Research Center, 1998.
- [DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998. To appear.
- [ESC] Extended Static Checking home page, Compaq Systems Research Center. On the Web at www.research.digital.com/SRC/esc/Esc.html.
- [GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.
- [Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–81, 1972.
- [Jac95] Daniel Jackson. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.
- [Jon91] H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM'91 Formal Software Development Methods, 4th International Symposium of VDM Europe, Volume 1: Conference Proceedings*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, October 1991.
- [Lam93] John Lampert. Typing the specialization interface. *ACM SIGPLAN Notices*, 28(10):201–214, October 1993. OOPSLA '93 conference proceedings.
- [LB97] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. Technical Report TR #97-19, Department of Computer Science, Iowa State University, September 1997.
- [Lea89] Gary Todd Leavens. *Verifying Object-Oriented Programs that Use Subtypes*. PhD thesis, MIT Laboratory for Computer Science, February 1989. Available as Technical Report MIT/LCS/TR-439.
- [Lea96] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, 1996.
- [Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.
- [Lei97] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from www.cs.indiana.edu/hyplan/pierce/fool/.
- [Lei98a] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. In Chris Hankin, editor, *Programming Languages and Systems: 7th European Symposium on Programming, ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*. Springer, April 1998.
- [Lei98b] K. Rustan M. Leino. Specifying the modification of extended state. In *The Fifth International Workshop on Foundations of Object-Oriented Languages*, January 1998. Proceedings available from www.pauillac.inria.fr/~remy/fool/program.html.
- [LH94] Kevin Lano and Howard Haughton, editors. *Object-Oriented Specification Case Studies*. The Object-Oriented Series. Prentice Hall, 1994.
- [LN98a] K. Rustan M. Leino and Greg Nelson. Abstraction and specification revisited. Internal manuscript KRML 71, Digital Equipment Corporation Systems Research Center. To appear as Compaq SRC Research Report 160, 1998.
- [LN98b] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In Kai Koskimies, editor, *Compiler Construction; Proceedings of the 7th International Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer, March 1998.

- [LS97] K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Note 1997-007, Digital Equipment Corporation Systems Research Center, April 1997.
- [LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
- [Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.
- [Nau94] David A. Naumann. Predicate transformer semantics of an Oberon-like language. In E.-R. Olderog, editor, *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods, and Calculi*, pages 467–487. Elsevier, June 1994.
- [PHM98] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods, PROCOMET '98*, pages 404–423. Chapman & Hall, 1998.
- [Sta97] Raymie Stata. Modularity in the presence of subclassing. Research Report 145, Digital Equipment Corporation Systems Research Center, April 1997.
- [Van94] Mark T. Vandervoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, Massachusetts Institute of Technology, February 1994. Available as Technical Report MIT/LCS/TR-598.

Separation and Information Hiding

Peter W. O’Hearn

Queen Mary
University of London

Hongseok Yang

Seoul National
University

John C. Reynolds

Carnegie Mellon
University

Abstract

We investigate proof rules for information hiding, using the recent formalism of separation logic. In essence, we use the separating conjunction to partition the internal resources of a module from those accessed by the module’s clients. The use of a logical connective gives rise to a form of dynamic partitioning, where we track the transfer of ownership of portions of heap storage between program components. It also enables us to enforce separation in the presence of mutable data structures with embedded addresses that may be aliased.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Program Verification—*class invariants*; D.3.3 [Programming Languages]: Language Constructs and Features—*modules, packages*

General Terms: Languages, Theory, Verification

Keywords: Separation Logic, Modularity, Resource Protection

1 Introduction

Modularity is a key concept which programmers wield in their struggle against the complexity of software systems. When a program is divided into conceptually distinct modules or components, each of which owns separate internal resources (such as storage), the effort required for understanding the program is decomposed into circumscribed, hopefully manageable, parts. And, if separation is correctly maintained, we can regard the internal resources of one module as hidden from its clients, which results in a narrowing of interface between program components. The flipside, of course, is that an ostensibly modular program organization is undermined when internal resources are accessed from outside a module.

It stands to reason that, when specifying and reasoning about programs, if we can keep track of the separation of resources between

program components, then the resultant decomposition of the specification and reasoning tasks should confer similar benefits. Unfortunately, most methods for specifying programs either severely restrict the programming model, by ruling out common programming features (so as to make the static enforcement of separation feasible), or they expose the internal resources of a module in its specification in order to preserve soundness.

Stated more plainly, information hiding should be the bedrock of modular reasoning, but it is difficult to support soundly, and this presents a great challenge for research in program logic.

To see why information hiding in specifications is desirable, suppose a program makes use of n different modules. It would be unfortunate if we had to thread descriptions of the internal resources of each module through steps when reasoning about the program. Even worse than the proof burden would be the additional annotation burden, if we had to complicate specifications of user procedures by including descriptions of the internal resources of all modules that might be accessed. A change to a module’s internal representation would necessitate altering the specifications of all other procedures that use it. The resulting breakdown of modularity would doom any aspiration to scalable specification and reasoning.

Mutable data structures with embedded addresses (pointers) have proven to be a particularly obstinate obstacle to modularity. The problem is that it is difficult to keep track of aliases, different copies of the same address, and so it is difficult to know when there are no pointers into the internals of a module. The purpose of this paper is to investigate proof rules for information hiding using separation logic, a recent formalism for reasoning about mutable data structures [36].

Our treatment draws on work of Hoare on proof rules for data abstraction and for shared-variable concurrency [16, 17, 18]. In Hoare’s approach each distinct module has an associated resource invariant, which describes its internal state, and scoping constraints are used to separate the resources of a module from those of client programs. We retain the resource invariants, and add a logical connective, the separating conjunction $*$, to provide a more flexible form of separation.

We begin in the next section by describing the memory model and the logic of pre- and post-conditions used in this work. We then describe our proof rules for information hiding, followed by two examples, one a simple memory manager module and the other a queue module. Both examples involve the phenomenon of *resource ownership transfer*, where the right to access a data structure transfers between a module and its clients. We work through proofs, and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL’04, January 14–16, 2004, Venice, Italy.

Copyright 2004 ACM 1-58113-729-X/04/0001 ...\$5.00

failed proofs, of client code as a way to illustrate the consequences of the proof rules.

After giving the positive examples we present a counterexample, which shows that our principal new proof rule, the hypothetical frame rule, is incompatible with the usual Hoare logic rule for conjunction; the new rule is thus unsound in models where commands denote relations, which validate conjunction. The problem is that the very features that allow us to treat ownership transfer lead to a subtle understanding where “Ownership is in the eye of the Asserter”. The remainder of the paper is occupied with a semantic analysis. This revolves around a notion of “precise” predicates, which are ones that unambiguously identify a portion of state. In essence, we ask the Asserter to be unambiguous when specifying which resource is owned; when this is the case, we find that our proof rules are sound.

Familiarity with the basics of separation logic, as presented in [36], would be helpful in reading the paper. We remind the reader in particular that the rules for disposing or dereferencing an address are such that it must be known to point to something (not be dangling) in the precondition for a rule to apply. For example, in the putative triple $\{\text{true}\}[x] := 7\{\text{??}\}$, where the contents of heap address x is mutated to 7, there is no assertion we can use in the postcondition to get a valid triple, because x might be dangling in a state satisfying the precondition. So, in order to obtain any postcondition for $[x] := 7$, the precondition must imply the assertion $x \mapsto * \text{true}$ that x is not dangling.

The local way of thinking encouraged by separation logic [26] is stretched by the approach to information hiding described here. We have found it useful to use a figurative language of “rights” when thinking about specifications, where a predicate p at a program point asserts that “I have the right to dereference the addresses in p here”.

1.1 Contextual Remarks

The link between modularity and information hiding was developed in papers of Hoare and Parnas in the early 1970s [16, 17, 28, 29]. Parnas emphasized that poor information distribution amongst components could lead to “almost invisible connections between supposedly independent modules”, and proposed that information hiding was a way to combat this problem. Hoare suggested using scoping restrictions to hide a particular kind of information, the internal state of a module, and showed how these restrictions could be used in concert with invariants to support proof rules that did not need to reveal the internal data of a module or component. These ideas influenced many subsequent language constructs and specification notations.

Most formal approaches to information hiding work by assuming a fixed, *a priori*, partitioning between program components, usually expressed using scoping restrictions, or typing, or simply using cartesian product of state spaces. In simple cases fixed partitioning can be used to protect internal resources from outside tampering. But in less simple situations, such as when data is referred to indirectly via addresses, or when resources dynamically transfer between program components, correct separation is more difficult to maintain. Such situations are especially common in low-level systems programs whose purpose is to provide flexible, shared access to system resources. They are also common in object-oriented programs. An unhappy consequence is that modular specification methods are lacking for widely-used imperative or object-oriented

programming languages, or even for many of the programming patterns commonly expressed in them.

The essential point is that fixed partitioning does not cope naturally with systems whose resource ownership or interconnection structure is changing over time. A good example is a resource management module, that provides primitives for allocating and deallocating resources, which are held in a local free list. A client program should not alter the free list, except through the provided primitives; for example, the client should not tie a cycle in the free list. In short, the free list is owned by the manager, and it is (intuitively) hidden from client programs. However, it is entirely possible for a client program to hold an alias to an element of the free list, after a deallocation operation is performed; intuitively, the “ownership” of a resource transfers from client to module on disposal, even if many aliases to the resource continue to be held by the client code. In a language that supports address arithmetic the potential difficulties are compounded: the client might intentionally or unintentionally obtain an address used in an internal representation, just by an arithmetic calculation.

A word of warning on our use of “module” before we continue: The concept of module we use is just a grouping of procedures that share some private state. The sense of “private” will not be determined statically, but will be the subject of specifications and proof rules. This allows us to approach modules where correct protection of module internals would be impossible to determine with a compile-time check in current programming languages. The approach in this paper might conceivably be used to analyze the information hiding in a language that provides an explicit module notation, but that is not our purpose here.

The point is that it is possible to program modules, in the sense of the word used by Parnas, whether or not one has a specific module construct at one’s disposal. For example, the pair of `malloc()` and `free()` in C, together with their shared free list, might be considered as a module, even though their correct usage is not guaranteed by C’s compile-time checking. Indeed, there is no existing programming language that correctly enforces information hiding of mutable data structures, largely because of the dynamic partitioning issue mentioned above, and this is an area where logical specifications are needed. We emphasize that the issue is not one of “safe” versus “unsafe” programming languages; for instance, middleware programs written in garbage-collected, safe languages, often perform explicit management of certain resources, and there also ownership transfer is essential to information hiding.

Similarly, although we do not consider the features of a full-blown object-oriented language, our techniques, and certainly our problems, seem to be relevant. Theories of objects have been developed that account for hiding in a purely functional context (e.g., [30]), but mutable structures with embedded addresses, or object id’s, are fundamental to object-oriented programming. A thoroughgoing theory should account for them directly, confronting the problems caused when there are potential aliases to the state used within an object.

These contextual remarks do not take into account some recent work that attempts to address the limitations of fixed partitioning and the difficulties of treating mutable data structures with embedded addresses. We will say more on some of the closely related work at the end of the paper.

2 The Storage Model

We consider a model where a heap is a finite partial function taking addresses to values:

$$H \stackrel{\text{def}}{=} \text{Addresses} \rightarrow_{\text{fin}} \text{Values}$$

This set has a partial commutative monoid structure, where the unit is the empty function and the partial combining operation

$$*: H \times H \rightharpoonup H$$

is the union of partial functions with disjoint domains. More formally, we say that $h_1 \# h_2$ holds for heaps h_1 and h_2 when $\text{dom}(h_1) \cap \text{dom}(h_2) = \{\}$. In that case, $h_1 * h_2$ denotes the combined heap $h_1 \cup h_2$. When $h_1 \# h_2$ fails, $h_1 * h_2$ is undefined. In particular, note that if $h = h_1 * h_2$ then we must have that $h_1 \# h_2$. The subheap order \leq is subset inclusion of partial functions.

We will work with a RAM model, where the addresses are natural numbers and the values are integers

$$\text{Addresses} \stackrel{\text{def}}{=} \{0, 1, 2, \dots\} \quad \text{Values} \stackrel{\text{def}}{=} \{\dots, -1, 0, 1, \dots\}$$

The results of this paper go through for other choices for Addresses and Values, and thus cover a number of other naturally occurring models, such as the cons cell model of [36] and the hierarchical memory model of [1]. Our results also apply to traditional Hoare logic, where there is no heap, by taking the trivial model where Addresses is empty (and Values non-empty).

A natural model of separation that is not an instance of the partial functions model construction above is the “trees with dangling pointers” model of [6]; it would be interesting to axiomatize the essentials of these separation models, by identifying a subclass of the partial monoid models of [32].

To interpret variables in the programming language and logic, the state has an additional component, the “stack”, which is a mapping from variables to values; a state is then a pair consisting of a stack and a heap:

$$S \stackrel{\text{def}}{=} \text{Variables} \rightarrow \text{Values} \quad \text{States} \stackrel{\text{def}}{=} S \times H.$$

We treat predicates semantically in this paper, so a predicate is just a set of states.

$$\text{Predicates} \stackrel{\text{def}}{=} \mathcal{P}(\text{States})$$

The powerset of states has the usual boolean algebra structure, where \wedge is intersection, \vee is union, \neg is complement, true is the set of all states, and false is the empty set of states. We use p, q, r , sometimes with subscripts and superscripts, to range over predicates. Besides the boolean connectives, we will need the lifting of $*$ from heaps to predicates:

$$p * q \stackrel{\text{def}}{=} \{(s, h) \mid \exists h_0, h_1. h = h_0 * h_1, \text{ and } (s, h_0) \in p, \text{ and } (s, h_1) \in q\}.$$

As a function on predicates we have a total map $*$ from Predicates \times Predicates to Predicates which, to the right of $\stackrel{\text{def}}{=}$, uses the partial map, $* : H \times H \rightharpoonup H$ in its definition. This overloading of $*$ will always be disambiguated by context. $*$ has a unit emp , the set $\{(s, []) \mid s \in S\}$ of states whose heap component is empty. It also has an implication adjoint $\text{--}*$, though that will play no role in the present paper. Note that emp is distinct from the empty set false of states.

We use $x \mapsto E$ to denote a predicate that consists of all pairs (s, h) where h is a singleton in which x points to the meaning of E :

$h(s(x)) = [[E]]s$. The points-to relation $x \mapsto E, F$ for binary cons cells is syntactic sugar for $(x \mapsto E) * (x + 1 \mapsto F)$. We will also use quantifiers and recursive definitions in examples in what should be a clear way.

The syntax for the programming language considered in this paper is given by the following grammar.

$$\begin{aligned} E &::= x, y, \dots \mid 0 \mid 1 \mid E + E \mid E \times E \mid E - E \\ B &::= \text{false} \mid B \Rightarrow B \mid E = E \mid E < E \\ C &::= x := E \mid x := [E] \mid [E] := E \mid x := \text{cons}(E, \dots, E) \\ &\quad \mid \text{dispose}(E) \mid \text{skip} \mid C; C \mid \text{if } B \text{ then } C \text{ else } C \\ &\quad \mid \text{while } B \text{ do } C \mid \text{letrec } k = C, \dots, k = C \text{ in } C \mid k \end{aligned}$$

For simplicity we consider parameterless procedures only. The extension to all first-order procedures raises no new difficulties, but lengthens the presentation. Higher-order features, on the other hand, are not straightforward. We assume that all the procedure identifiers are distinct in any `letrec` declaration. When procedure declarations do not have recursive calls, we write `let` $k_1 = C_1, \dots, k_n = C_n$ in C to indicate this.

The command $x := \text{cons}(E_1, \dots, E_n)$ allocates n consecutive cells, initializes them with the values of E_1, \dots, E_n , and stores the address of the first cell in x . We could also consider a command for variable-length allocation. The contents of an address E can be read and stored in x by $x := [E]$, or can be modified by $[E] := F$. The command `dispose(E)` deallocates the address E . In $x := [E]$, $[E] := F$ and `dispose(E)`, the expression E can be an arbitrary arithmetic expression; so, this language allows address arithmetic.

This inclusion of address arithmetic does not represent a general commitment to it on our part, but rather underlines the point that our methods do not rely on ruling it out. In examples it is often clearer to use a field-selection notation rather than arithmetic, and for this we use the following syntactic sugar:

$$E.i := F \stackrel{\text{def}}{=} [E + i - 1] := F \quad x := E.i \stackrel{\text{def}}{=} x := [E + i - 1].$$

Each command denotes a (nondeterministic) state transformer that faults when heap storage is accessed illegally, and each expression determines a (heap independent) function from stacks to values. The semantics will be given in Section 7.

3 Proof System

The form of judgment we use is the sequent

$$\Gamma \vdash \{p\}C\{q\}$$

which states that command C satisfies its Hoare triple, under certain hypotheses. Hypotheses are given by the grammar

$$\Gamma ::= \varepsilon \mid \{p\}k\{q\}[X], \Gamma$$

subject to the constraint that no procedure identifier k appears twice. An assumption $\{p\}k\{q\}[X]$ requires k to denote a command that modifies only the variables appearing in set X and that satisfies the indicated triple.

3.1 Proof Rules for Information Hiding

We begin with a special-case, programmer-friendly, proof rule, that is a consequence of a more fundamental, logician-friendly, rule to be described later.

Modular Non-Recursive Procedure Declaration Rule

$$\frac{\Gamma \vdash \{p_1 * r\} C_1 \{q_1 * r\} \\ \vdots \\ \Gamma \vdash \{p_n * r\} C_n \{q_n * r\} \\ \Gamma, \{p_1\} k_1 \{q_1\} [X_1], \dots, \{p_n\} k_n \{q_n\} [X_n] \vdash \{p\} C \{q\}}{\Gamma \vdash \{p * r\} \text{let } k_1 = C_1, \dots, k_n = C_n \text{ in } C \{q * r\}}$$

In this rule k_1, \dots, k_n is a grouping of procedures that share private state described by resource invariant r . In a resource management module, the k_i would be operations for allocating and freeing resources, and r would describe unallocated resources (perhaps held in a free list). The rule distinguishes two views of such a module. When reasoning about the client code C , we ignore the invariant and its area of storage; reasoning is done in the context of *interface specifications* $\{p_i\} k_i \{q_i\}$ that do not mention r . The perspective is different from inside the module; the implementations C_i operate on a larger state than that presented to the client, and verifications are performed in the presence of the resource invariant. The two views, module and client, are tied up in the conclusion of the rule.

The modular procedure rule is subject to variable conditions: we require a set Y (of “private” variables), and the conditions are

- C does not modify variables in r , except through using k_1, \dots, k_n ;
- Y is disjoint from p, q, C and the context “ $\Gamma, \{p_1\} k_1 \{q_1\} [X_1], \dots, \{p_n\} k_n \{q_n\} [X_n]$ ”;
- C_i only modifies variables in X_i, Y .

The idea behind these conditions is that we must be sure that client code does not alter variables used within a module, but we must also allow some overlap in variables to treat various examples. A rigorous formulation of what these conditions mean has been placed in an appendix at the end of the paper. We will continue to state the necessary side conditions as we present our proof rules, but there will be little harm if the reader skates over them, or understands them in an intuitive way, while reading the paper. We only stress that the modifies clauses refer exclusively to the stack, where the new part of the paper involves the heap, and $*$.

It is also possible to consider initialization and finalization code. For instance, if, in addition to the premises of the modular procedure rule, we have $\Gamma \vdash \{p\} \text{init}\{p * r\}$ and $\Gamma \vdash \{q * r\} \text{final}\{q\}$, then we can obtain

$$\Gamma \vdash \{p\} \text{ init}; (\text{let } k_1 = C_1, \dots, k_n = C_n \text{ in } C); \text{ final } \{q\}.$$

In our examples we will not consider initialization or finalization since they present no special logical difficulties.

In the modular procedure rule, the proof of $\{p\} C \{q\}$ about the client in the premises can be used with *any* resource invariant r . As a result, this reasoning does not need to be repeated when a module representation is altered, as long as the alteration continues to satisfy the interface specifications $\{p_i\} k_i \{q_i\}$. This addresses one of the points about reasoning that survives local changes discussed in the Introduction.

However, the choice of invariant r is not specified by programming language syntax $\text{let } k_1 = C_1, \dots, k_n = C_n \text{ in } C$ in the modular procedure rule. In this it is similar to the usual partial correctness rule for while loops, which depends on the choice of a loop invariant. It will be convenient to consider an annotation notation that specifies the invariant, and the interface specifications $\{p_i\} k_i \{q_i\}$, as a directive on how to apply the modular procedure rule; this is by

Interface Specifications

$$\{p_1\} k_1 \{q_1\} [X_1], \dots, \{p_n\} k_n \{q_n\} [X_n]$$

Resource Invariant: r

Private Variables: Y

Internal Implementations

$$C_1, \dots, C_n$$

Table 1. Module Specification Format

analogy with the use of loop invariant annotations as directives to a verification condition generator.

We will use the format for module specifications in Table 1. This instructs us to apply the modular procedure rule in a particular way, to prove

$$\Gamma, \text{Interface Specifications} \vdash \{p\} C \{q\}$$

for client code C , and to prove $\Gamma \vdash \{p_i * r\} C_i \{q_i * r\}$ for the bodies. We emphasize that this module format is not officially part of our programming language or even our logic; however, its role as a directive on how to apply the modular procedure rule in examples will, we hope, be clear.

The modular procedure rule can be derived from a standard rule for parameterless procedure declarations, and the following more basic rule.

Hypothetical Frame Rule

$$\frac{\Gamma, \{p_i\} k_i \{q_i\} [X_i]_{(\text{for } i \leq n)} \vdash \{p\} C \{q\} \quad \Gamma, \{p_i * r\} k_i \{q_i * r\} [X_i, Y]_{(\text{for } i \leq n)} \vdash \{p * r\} C \{q * r\}}{\Gamma, \{p\} k \{q\} [X_1, \dots, X_n, Y] \vdash \{p\} C \{q\}}$$

where

- C does not modify variables in r , except through using k_1, \dots, k_n ; and
- Y is disjoint from p, q, C , and the context “ $\Gamma, \{p_1\} k_1 \{q_1\} [X_1], \dots, \{p_n\} k_n \{q_n\} [X_n]$ ”.

The notation $\{p_i\} k_i \{q_i\} [X_i]_{(\text{for } i \leq n)}$ in the rule is a shorthand for $\{p_1\} k_1 \{q_1\} [X_1], \dots, \{p_n\} k_n \{q_n\} [X_n]$, and similarly for $\{p_i * r\} k_i \{q_i * r\} [X_i, Y]_{(\text{for } i \leq n)}$. In examples we will use the modular procedure rule, but will phrase our theoretical results in terms of the hypothetical frame rule.

The hypothetical frame rule is so named because of its relation to the ordinary frame rule from [20, 26]. The hypothetical rule allows us to place invariants on the hypotheses as well as the conclusion of sequents, whereas the ordinary rule includes invariants on the conclusion alone. (The ordinary frame rule is thus a special case of the hypothetical rule, where $n = 0$.)

3.2 Other Proof Rules

We have standard Hoare logic rules for various constructs, along with the rule of consequence.

$$\frac{p \Rightarrow p' \quad \Gamma \vdash \{p'\} C \{q'\} \quad q' \Rightarrow q}{\Gamma, \{p\} k \{q\} [X] \vdash \{p\} k \{q\}}$$

$$\frac{\Gamma \vdash \{p \wedge B\} C \{p\} \quad \Gamma \vdash \{p\} C_1 \{q\} \quad \Gamma \vdash \{q\} C_2 \{r\}}{\Gamma \vdash \{p\} \text{while } B \text{ do } C \{p \wedge \neg B\}}$$

$$\frac{\Gamma \vdash \{p \wedge B\} C \{q\} \quad \Gamma \vdash \{p \wedge \neg B\} C' \{q\}}{\Gamma \vdash \{p\} \text{if } B \text{ then } C \text{ else } C' \{q\}}$$

In addition, we allow for the context Γ to be permuted.

The rule for possibly recursive procedure declarations uses the procedure specifications in proofs of the bodies:

$$\frac{\begin{array}{c} \Gamma, \{p_1\}k_1\{q_1\}[X_1], \dots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p_1\}C_1\{q_1\} \\ \vdots \\ \Gamma, \{p_1\}k_1\{q_1\}[X_1], \dots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p_n\}C_n\{q_n\} \\ \Gamma, \{p_1\}k_1\{q_1\}[X_1], \dots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\} \end{array}}{\Gamma \vdash \{p\}\text{letrec } k_1 = C_1, \dots, k_n = C_n \text{ in } C\{q\}}$$

where

- C_i only modifies variables in X_i .

In case none of the k_i are free in the C_j we can get a simpler rule, where the $\{p_i\}k_i\{q_i\}[X_i]$ hypotheses are omitted from the sequents for the C_j . Using `let` rather than `letrec` to indicate the case where a procedure declaration happens to have no recursive instances, we can derive the modular non-recursive procedure declaration rule of the previous section from the hypothetical frame rule and the standard procedure rule just given. We can also derive a modular rule for recursive declarations.

The ordinary frame rule is

$$\frac{\Gamma \vdash \{p\}C\{q\}}{\Gamma \vdash \{p * r\}C\{q * r\}}$$

where

- C does not modify any variables in r .

This is a special case of the hypothetical rule, but we state it separately because the ordinary rule will be used without restriction, while we will place restrictions on the hypothetical rule.

One rule of Hoare logic, which is sometimes not included explicitly in proof systems, is the conjunction rule.

$$\frac{\Gamma \vdash \{p\}C\{q\} \quad \Gamma \vdash \{p'\}C\{q'\}}{\Gamma \vdash \{p \wedge p'\}C\{q \wedge q'\}}$$

The conjunction rule is often excluded because it is an example of an *admissible* rule: one can (usually) prove a metatheorem, which says that if the premises are derivable then so is the conclusion. However, it is not an example of a *derived* rule: one cannot construct a generic derivation, in the logic, of the conclusion from the premises. We will see in Section 6 that the hypothetical frame rule can affect the admissible status of the conjunction rule.

Finally, we have axioms for the basic commands, where x, m, n are assumed to be distinct variables.

$$\begin{aligned} & \Gamma \vdash \{E \mapsto -\}[E] := F \{E \mapsto F\} \\ & \Gamma \vdash \{E \mapsto -\} \text{dispose}(E) \{\text{emp}\} \\ & \Gamma \vdash \left\{ \begin{array}{l} x = m \\ \wedge \text{emp} \end{array} \right\} x := \text{cons}(E_1, \dots, E_k) \{x \mapsto E_1[m/x], \dots, E_k[m/x]\} \\ & \Gamma \vdash \{x = n \wedge \text{emp}\} x := E \{x = (E[n/x]) \wedge \text{emp}\} \\ & \Gamma \vdash \{E \mapsto n \wedge x = m\} x := [E] \{x = n \wedge E[m/x] \mapsto n\} \end{aligned}$$

These axioms describe the effect of each command on only one, or sometimes no, heap cells. Typically, their effects can be extended using the frame rule: for example, we can infer $\{(x \mapsto 3) * (y \mapsto 4)\}[x] := 7\{(x \mapsto 7) * (y \mapsto 4)\}$ by choosing $y \mapsto 4$ as the invariant in the frame rule.

Interface Specifications

$$\begin{array}{l} \{\text{emp}\} \text{alloc}\{x \mapsto -, -\}[x] \\ \{x \mapsto -, -\} \text{free}\{\text{emp}\} [] \end{array}$$

Resource Invariant: $list(f)$

Private Variables: f

Internal Implementations

$$\begin{array}{ll} \text{if } f = \text{nil} \text{ then } x := \text{cons}(-, -) & (\text{code for alloc}) \\ \text{else } x := f; f := x.2; & \\ x.2 := f; f := x; & (\text{code for free}) \end{array}$$

Table 2. Memory Manager Module

4 A Memory Manager

We consider an extended example, of an idealized memory manager that doles out memory in chunks of size two. The specifications and code are given in Table 2.

The internal representation of the manager maintains a free list, which is a singly-linked list of binary cons cells. The free list is pointed to by f , and the predicate $list(f)$ is the representation invariant, where

$$list(f) \stackrel{\text{def}}{\iff} (f = \text{nil} \wedge \text{emp}) \vee (\exists g. f \mapsto -, g * list(g))$$

This predicate says that f points to a linked list (and that there are no other cells in storage), but it does not say what elements are in the head components.

For the implementation of `alloc`, the manager places into x the address of the first element of the free list, if the list is nonempty. In case the list is empty the manager calls the built-in allocator `cons` to get an extra element. The interaction between `alloc` and `cons` is a microscopic idealization of the treatment of `malloc` in Section 8.7 of [22]. There, `malloc` manages a free list but, occasionally, it calls a system routine `sbrk` to request additional memory. Besides fixed versus variable sized allocation, the main difference is that we assume that `cons` always succeeds, while `sbrk` might fail (return an error code) if there is no extra memory to be given to `malloc`. We use this simple manager because to use a more complex one would not add anything to the points made in this section.

When a user program gives a cell back to the memory manager it is put on the front of the free list; there is no need for interaction with a system routine here.

The form of the interface specifications are examples of the local way of thinking encouraged by separation logic; they refer to small pieces of storage. It is important to appreciate the interaction between local and more global perspectives in these assertions. For example, in the implementation of `free` in Table 2 the variable x contains the same address after the operation completes as it did before, and the address continues to be in the domain of the global program heap. The use of `emp` in the postcondition of `free` does not mean that the global heap is now empty, but rather it implies that the knowledge that x points to something is given up in the postcondition. We say intuitively that `free` transfers ownership to the manager, where ownership confers the right to dereference.

It is interesting to see how transfer works logically, by considering a proof outline for the implementation of `free`.

```

{list(f) * (x ↦ -, -)}
x.2 := f;
{list(f) * (x ↦ -, f)}
{list(x)}
f := x;
{list(f)}
{list(f) * emp}

```

The most important step is the middle application of the rule of consequence. At that point we still have the original resource invariant $\text{list}(f)$ and the knowledge that x points to something, separately. But since the second field of what x points to holds f , we can obtain $\text{list}(x)$ as a consequence. It is at this point in the proof that the original free list and the additional element x are bundled together; the final statement simply lets f refer to this bundled information.

A similar point can be made about how `alloc` effects a transfer from the module to the client.

We now give several examples from the client perspective. Each proof, or attempted proof, is done in the context of the interface specifications of `alloc` and `free`.

The first example is for inserting an element into the middle of a linked list.

```

{(y ↦ a, z) * (z ↦ c, w)}
alloc;
{(y ↦ a, z) * (z ↦ c, w) * (x ↦ -, -)}
{(y ↦ a, z) * (x ↦ -, -) * (z ↦ c, w)}
x.2 := z; x.1 := b; y.2 := x
{(y ↦ a, x) * (x ↦ b, z) * (z ↦ c, w)}

```

Here, in the step for `alloc` we use the interface specification, together with the ordinary frame rule.

If we did not have the modular procedure rule we could still verify this code, by threading the free list through and changing the interface specification. That is, the interface specifications would become

```

{list(f)} alloc {list(f) * x ↦ -, -}
{list(f) * x ↦ -} free {list(f)}

```

thus exposing the free list, and the proof would be

```

{(y ↦ a, z) * (z ↦ c, w) * list(f)}
alloc;
{(y ↦ a, z) * (z ↦ c, w) * (x ↦ -, -) * list(f)}
{(y ↦ a, z) * (x ↦ -, -) * (z ↦ c, w) * list(f)}
x.2 := z; x.1 := b; y.2 := x
{(y ↦ a, x) * (x ↦ b, z) * (z ↦ c, w) * list(f)}.

```

Although technically correct, this inclusion of the free list in the proof of the client is an example of the breakdown of modularity described in the Introduction.

One might wonder whether this hiding of invariants could be viewed as a simple matter of syntactic sugar, instead of being the subject of a proof rule. We return to this point in Section 6.

We can similarly reason about deletion from the middle of a linked list, but it is more interesting to attempt to delete wrongly.

```

{(y ↦ a, x) * (x ↦ b, z) * (z ↦ c, w)}
free;

```

```

{(y ↦ a, x) * (z ↦ c, w)}
y := x.2;
{???
}
```

This verification cannot be completed, because after doing the `free` operation the client has given up the right to dereference x .

This is a very simple example of the relation between ownership transfer and aliasing; after the `free` operation x and f are aliases in the global state, and the incorrect use of the alias by the client has been rightly precluded by the proof rules. (A more positive example of aliasing, which incidentally would not be amenable to unique-reference disciplines, would be a program to dispose nodes in a graph.)

Similarly, suppose the client tried to corrupt the manager, by sneakily tying a cycle in the free list.

```
{emp} alloc; free; x.2 := x {???
```

Once again, there is no assertion we can find to fill in the $???$, because after the `free` statement the client has given up the right to dereference x (`emp` will hold at this program point). And, this protection has nothing to do with the fact that knotting the free list contradicts the resource invariant. For, suppose the statement $x.2 := x$ was replaced by $x.1 := x$. Then the final assignment in this sequence would not contradict the resource invariant, when viewed from the perspective of the system's global state, because the $\text{list}(f)$ predicate is relaxed about what values are in head components. However, from the point of view of the interface specifications, the client has given up the right to dereference even the first component of x . Thus, separation prevents the client from accessing the internal storage of the module in any way whatsoever.

Finally, it is worth emphasizing that this use of $*$ to enforce separation provides protection even in the presence of address arithmetic which, if used wrongly, can wreak havoc with data abstractions. Suppose the client tries to access some memory address, which might or might not be in the free list, using $[42] := 7$. Then, for this statement to get past the proof rules, the client must have the right to dereference 42, and therefore 42 cannot be in the free list (by separation). That is, we have two cases

```
{42 ↦ - * p} [42] := 7; alloc {42 ↦ 7 * p * x ↦ -, -}
```

and

```
{p} [42] := 7; {???
} alloc {???
```

where p does not imply that 42 is in the domain of its heap. In the first case the client has used address arithmetic correctly, and the $42 \rightarrow -$ in the precondition ensures that 42 is not one of the cells in the free list. In the second case the client uses address arithmetic potentially incorrectly, and the code might indeed corrupt the free list, but the code is (in the first step) blocked by the proof rules.

5 The Eye of the Asserter

In Table 3 we give a queue module. In the specification we use a predicate $\text{listseg}(x, \alpha, y)$ which says that there is an acyclic linked list from x to y has the sequence α in its head components. The variable Q denotes the sequence of values currently held in the queue; it is present in the resource invariant, as well as in the interface specifications. (Technically, we would have to ensure that the variable Q was added to the s component of our semantics.) This exposing of “abstract” variables is standard in module specifications, as is the inclusion of assignment statements involving abstract variables

Interface Specifications

$$\begin{aligned} \{Q = \alpha \wedge z = n \wedge P(z)\} \text{enq } \{Q = \alpha \cdot \langle n \rangle \wedge \text{emp}\} [Q] \\ \{Q = \langle m \rangle \cdot \alpha \wedge \text{emp}\} \text{deq } \{Q = \alpha \wedge z = m \wedge P(z)\} [Q, z] \\ \{\text{emp}\} \text{isempty? } \{(w = (Q = \varepsilon)) \wedge \text{emp}\} [w] \end{aligned}$$

Resource Invariant: $\text{listseg}(x, Q, y) * (y \mapsto -, -)$

Private Variables: x, y, t

listseg Predicate Definition

$$\begin{aligned} \text{listseg}(x, \alpha, y) &\stackrel{\text{def}}{\iff} \\ \text{if } x = y \text{ then } (\alpha = [] \wedge \text{emp}) \\ \text{else } (\exists v, z, \alpha'. (\alpha = \langle v \rangle \cdot \alpha' \wedge x \mapsto v, z) * P(v) \\ &\quad * \text{listseg}(z, \alpha', y)) \end{aligned}$$

Internal Implementations

$$\begin{aligned} Q := Q \cdot \langle z \rangle; &\quad (\text{code for enq}) \\ t := \text{cons}(-, -); y.1 := z; y.2 := t; y := t \\ Q := \text{cdr}(Q); &\quad (\text{code for deq}) \\ z := x.1; t := x; x := x.2; \text{dispose}(t) \\ w := (x = y) &\quad (\text{code for isempty?}) \end{aligned}$$

Table 3. Queue Module, Parametric in $P(v)$

whose only purpose is to enable the specification to work.

This queue module keeps a sentinel at the end of its internal list, as is indicated by $(y \mapsto -, -)$ in the resource invariant. The sentinel does not hold any value in the queue, but reserves storage for a new value.

An additional feature of the treatment of queues is the predicate P , which is required to hold for each element of the sequence α . By instantiating P in various ways we obtain versions of the queue module that transfer different amounts of storage.

- $P(v) = \text{emp}$: plain values are transferred in and out of the queue, and no storage is transferred with any of these values;
- $P(v) = v \mapsto -, -$: binary cons cells, and ownership of the storage associated with them, are transferred in and out of the queue;
- $P(v) = \text{list}(v)$: linked lists, and ownership of the storage associated with them, are transferred in and out of the queue.

To illustrate the difference between these cases, consider the following attempted proof steps in client code.

$$\begin{aligned} \{Q = \langle n \rangle \cdot \alpha \wedge \text{emp}\} \\ \text{deq} \\ \{Q = \alpha \wedge z = n \wedge P(z)\} \\ z.1 := 42 \\ \{???\} \end{aligned}$$

In case $P(v)$ is either emp or $\text{list}(v)$ we cannot fill in $???$ because we do not have the right to dereference z in the precondition of $z.1 := 42$. However, if $P(v)$ is $v \mapsto -, -$ then we will have this right, and a valid postcondition is $(Q = \alpha \wedge z = n \wedge z \mapsto 42, -)$. Conversely, if we replace $z.1 := 42$ by code that traverses a linked list then the third definition of $P(v)$ will enable a verification to go through, where the other two will not.

On the other hand there is no operational distinction between these three cases: the queue code just copies values.

The upshot of this discussion is that the idea of ownership transfer we have alluded to is not determined by instructions in the programming language alone. Just what storage is, or is not, transferred depends on which definition of P we choose. And this choice depends on what we want to prove.

This phenomenon, where ‘‘Ownership is in the eye of the Asserter’’, can take some getting used to at first. One might feel ownership transfer might be made an explicit operation in the programming language. In some cases such a programming practice would be useful, but the simple fact is that in real programs the amount of resource transferred is not always determined operationally; rather, there is an understanding between a module writer, and programmers of client code. For example, when you call `malloc()` you just receive an address. The implementation of `malloc()` does not include explicit statements that transfer each of several cells to its caller, but the caller understands that ownership of several cells comes with the single address it receives.

6 A Conundrum

In the following 0 is the assertion emp that the heap is empty, and 1 says that it has precisely one active cell, say x (so 1 is $x \mapsto -$).

Consider the following instance of the hypothetical frame rule, where true is chosen as the invariant:

$$\frac{\{0 \vee 1\}k\{0\}[] \vdash \{1\}k\{\text{false}\}}{\{(0 \vee 1) * \text{true}\}k\{0 * \text{true}\}[] \vdash \{1 * \text{true}\}k\{\text{false} * \text{true}\}}$$

The conclusion is definitely false in any sensible semantics of sequents. For example, if k denotes the do-nothing command, `skip`, then the antecedent holds, but the consequent does not.

However, we can derive the sequent in the premise:

$$\frac{\begin{array}{c} \{0 \vee 1\}k\{0\} \\ \{1\}k\{0\} \end{array} \text{Consequence} \quad \frac{\begin{array}{c} \{0 \vee 1\}k\{0\} \\ \{0\}k\{0\} \end{array} \text{Consequence} \quad \frac{\begin{array}{c} \{0 * 1\}k\{0 * 1\} \\ \{1\}k\{1\} \end{array} \text{Consequence}}{\{1 \wedge 1\}k\{1 \wedge 0\}} \text{Conjunction}}{\{1\}k\{\text{false}\}} \text{Consequence}$$

This shows that we cannot have all of: the usual rule of consequence, the ordinary frame rule, the conjunction rule, and the hypothetical frame rule. It also shows that the idea of treating information hiding as syntactic sugar for proof and specification forms should be approached with caution: one needs to be careful that introduced sugar does not interact badly with expected rules, in a way that contradicts them.

The counterexample can also be presented as a module, and can be used to show a similar problem with the modular procedure rule.

Given this counterexample, the question is where to place the blame. There are several possibilities.

1. The specification $\{0 \vee 1\}k\{0\}$. This is an unusual specification, since in the programming languages we have been using there is no way to branch on whether the heap is empty.
2. The invariant true . Intuitively, a resource invariant should precisely identify an unambiguous area of storage, that owned by a module. The invariant $\text{list}(f)$ in the memory manager is unambiguous in this sense, where true is perhaps not.

3. One of the rules of conjunction, consequence, or the ordinary frame rule.

We pursue the first two options in the remainder of the paper, by defining a model of the programming language and investigating a notion of precise predicate.

7 A Denotational Model

Until now in work on separation logic we have used operational semantics, but in this paper we use a denotational semantics. By using denotational semantics we will be able to reduce the truth of a sequent $\Gamma \vdash \{p\}C\{q\}$ to the truth of a single semantic triple $\{p\}\llbracket C \rrbracket \eta\{q\}$ where η maps each procedure identifier in Γ to a “greatest” or “most general” relation satisfying it. In the case of the hypothetical frame rule, we will be able to compare two denotations of the same command for particular instantiations of the procedure identifiers, rather than having to quantify over all possible instantiations. Our choice to use denotational semantics here is entirely pragmatic: The greatest relation is not always definable by a program, but the ability to refer to it leads to significant simplifications in proofs about the semantics.

Recall that a state consists of a pair, of a stack and a heap. A command is interpreted as a binary relation

$$\text{States} \leftrightarrow \text{States} \cup \{\text{fault}\}$$

that relates an input state to possible output states, or a special output, `fault`, which indicates an attempted access of an address not in the domain of the heap. In fact, because we use a fault-avoiding interpretation of Hoare triples, it would be possible to use the domain

$$\text{States} \rightarrow \mathcal{P}(\text{States}) \cup \{\text{fault}\}$$

instead. Using the more general domain lets us see clearly that if a command nondeterministically chooses between `fault` and some state, then the possibility of faulting will mean that the command is not well specified according to the semantics of triples. This is not an essential point; the more constrained domain could be used without affecting any of our results.

This domain of relations is inappropriate for total correctness because it does not include a specific result for non-termination, so that our semantics will not distinguish a command C from one that nondeterministically chooses C or divergence.

We will not consider all relations, but rather only those that validate the locality properties of the (ordinary) frame rule. We say that a relation $c : \text{States} \leftrightarrow \text{States} \cup \{\text{fault}\}$ is *safe* at a state (s, h) when $\neg((s, h)[c] \text{fault})$. We just list the properties here, and refer the reader to [39] for further explanation of them. The locality properties are:

1. **Safety Monotonicity:** for all states (s, h) and heaps h_1 such that $h \# h_1$, if c is safe at (s, h) , it is also safe at $(s, h * h_1)$.
2. **Frame Property:** for all states (s, h) and heaps h_1 such that $h \# h_1$, if c is safe at (s, h) and $(s, h * h_1)[c](s', h')$, then there is a subheap $h'_0 \leq h'$ such that

$$h'_0 \# h_1, h'_0 * h_1 = h', \text{ and } (s, h)[c](s', h'_0).$$

Commands will be interpreted using the following domain.

The poset LRel of “local relations” is the set of all relations c satisfying the safety monotonicity and frame

properties, ordered by subset inclusion.

LEMMA 1. *LRel is a chain-complete partial order with a least element. The least element is the empty relation, and the least upper bound of a chain is given by the union of all the relations in the chain.*

The meaning of a command is given in the context of an environment η , that maps procedure identifiers to relations in LRel .

$$\eta \in \text{Procds} \rightarrow \text{LRel} \quad \llbracket C \rrbracket \eta \in \text{LRel}$$

The semantics of expressions depends only on the stack

$$\llbracket E \rrbracket s \in \text{Ints} \quad \llbracket B \rrbracket s \in \{\text{true}, \text{false}\} \quad (\text{where } s \in S).$$

The valuations are standard and omitted.

Selected valuations for commands are in Table 4. The main point to note is the treatment of `fault`. We have included only the basic commands and sequential composition. The interpretation of conditionals is as usual, a procedure call applies the environment to the corresponding variable, and while loops and `letrec` receive standard least fixed-point interpretations, which are guaranteed to exist by Lemma 1.

LEMMA 2. *For each command C , $\llbracket C \rrbracket$ is well-defined: for all environments η , $\llbracket C \rrbracket \eta$ is in LRel , and $\llbracket C \rrbracket \eta$ is continuous in η when environments are ordered pointwise.*

It is entertaining to see the nondeterminism at work in the semantics of `cons` in this model. In particular, since we are aiming for partial correctness, the semantics does not record whether a command terminates or not; for instance, $x := 1; y := 1$ has the same denotation as a command that nondeterministically picks either $x := 1; y := 1$ or divergence. Such a nondeterministic command can be expressed in our language as

$$\begin{aligned} x &:= \text{cons}(0); \text{dispose}(x); y := \text{cons}(0); \text{dispose}(y); \\ \text{if } (x = y) \text{ then } (x := 1; y := 1) \\ &\quad \text{else (while } (x = x) \text{ skip)} \end{aligned}$$

The reader may enjoy verifying that this is indeed equivalent to $x := 1; y := 1$ in the model.

8 Semantics of Sequents

In this section we give a semantics where a sequent

$$\Gamma \vdash \{p\}C\{q\}$$

says that if every specification in Γ is true of an environment, then so is $\{p\}C\{q\}$.

To interpret sequents we define semantic cousins of the modifies clauses and Hoare triples. If $c \in \text{LRel}$ is a relation then

- $\text{modifies}(c, X)$ holds if and only if whenever $y \notin X$ and $(s, h)[c](s', h')$, we have that $s(y) = s'(y)$.
- $\{p\}c\{q\}$ holds if and only if for all states (s, h) in p ,
 1. $\neg((s, h)[c] \text{fault})$; and
 2. if $(s, h)[c](s', h')$ then state (s', h') is in q .

Now we can define the semantics: A sequent

$$\{p_1\}k_1\{q_1\}[X_1] \dots, \{p_n\}k_n\{q_n\}[X_n] \vdash \{p\}C\{q\}$$

holds if and only if

for $(s, h) \in \text{States}$ and $a \in \text{States} \cup \{\text{fault}\}$,

$$\begin{aligned}
(s, h)[[x := E]\eta]a &\iff a = (s[x \mapsto [E]s], h) \\
(s, h)[[x := \text{cons}(E_1, \dots, E_n)]\eta]a &\iff \exists m. (m, \dots, m+n-1 \not\in \text{dom}(h)) \\
&\quad \wedge (a = (s[x \mapsto m], h * [m \mapsto [E_1]s, \dots, m+n-1 \mapsto [E_n]s])) \\
(s, h)[[x := [E]]\eta]a &\iff \text{if } [[E]]s \in \text{dom}(h) \text{ then } a = (s[x \mapsto h([[E]]s)], h) \text{ else } a = \text{fault} \\
(s, h)[[[E]] := F]\eta]a &\iff \text{if } [[E]]s \in \text{dom}(h) \text{ then } a = (s, h[[E]s \mapsto [F]s]) \text{ else } a = \text{fault} \\
(s, h)[[\text{dispose}(E)]\eta]a &\iff \text{if } [[E]]s \in \text{dom}(h) \\
&\quad \text{then } a = (s, h') \text{ for } h' \text{ s.t. } h' * ([E]s \mapsto h([E]s)) = h \\
&\quad \text{else } a = \text{fault} \\
(s, h)[[C_1; C_2]\eta]a &\iff (\exists (s', h'). (s, h)[[C_1]\eta] (s', h') \wedge (s', h')[[C_2]\eta] a) \vee ((s, h)[[C_1]\eta] \text{ fault} \wedge a = \text{fault})
\end{aligned}$$

where $\text{fix } f$ gives the least fixed-point of f , and $\text{seq}(c_1, c_2)$, $b \rightsquigarrow c_1; c_2$ and d_1, \dots, d_n are defined as follows:

$$\begin{aligned}
(s, h)[\text{seq}(c_1, c_2)]a &\iff (\exists (s', h'). (s, h)[c_1](s', h') \wedge (s', h')[c_2]a) \vee ((s, h)[c_1] \text{ fault} \wedge a = \text{fault}) \\
(s, h)[b \rightsquigarrow c_1; c_2]a &\iff \text{if } b(s) = \text{true} \text{ then } (s, h)[c_1]a \text{ else } (s, h)[c_2]a \\
(d_1, \dots, d_n) &= \text{fix}(\lambda d_1, \dots, d_n \in \text{LRel}^n. (F_1, \dots, F_n)) \quad (\text{where } F_i = [[C_i]]\eta[k_1 \mapsto d_1, \dots, k_n \mapsto d_n])
\end{aligned}$$

Table 4. Selected Valuations

for all environments η , if both $\{p_i\}\eta(k_i)\{q_i\}$ and $\text{modifies}(\eta(k_i), X_i)$ hold for all $1 \leq i \leq n$, the triple $\{p\}([[C]]\eta)\{q\}$ also holds.

9 Precise Predicates

We know from the counterexample in Section 6 that we must restrict the hypothetical frame rule in some way, if it is to be used with the standard semantics. Before describing the restriction, let us retrace some of our steps. We had a situation where ownership could transfer between a module and a client, which made essential use of the dynamic nature of $*$. But we had also got to a position where ownership is determined by what the Asserter asserts, and this put us in a bind: when the Asserter does not precisely specify what storage is owned, different splittings can be chosen at different times using the nondeterministic semantics of $*$; this fools the hypothetical frame rule. It is perhaps fortuitous that the nondeterminism in $*$ has not gotten us into trouble in separation logic before now. A way out of this problem is to insist that the Asserter precisely nail down the storage that he or she is talking about.

A predicate p is *precise* if and only if for all states (s, h) , there is at most one subheap h_p of h for which $(s, h_p) \in p$.

Intuitively, this definition says that for each state (s, h) , a precise predicate unambiguously specifies the portion of the heap h that is relevant to the predicate. Formulae that describe data structures are often precise. Indeed, the definition might be viewed as a formalization of a point of view stressed by Richard Bornat, that for practically any data structure one can write a formula or program that searches through a heap and picks out the relevant cells. Bornat used this idea of reading out the relevant cells in order to express spatial separation in traditional Hoare logic [4].

An example of a precise predicate is the following one for list segments:

$$\text{listseg}(x, y) \stackrel{\text{def}}{\iff} (x = y \wedge \text{emp}) \vee (x \neq y \wedge \exists z. (x \mapsto z) * \text{listseg}(z, y))$$

This predicate is true when the heap contains a non-circular linked list (and nothing else), which starts from the cell x and ends with y . Note that because of $x \neq y$ in the second disjunct, the predicate

$\text{listseg}(x, y)$ says that if x and y have the same value in a state (s, h) , the heap h must be empty. If we had left $x \neq y$ out of the second disjunct, then $\text{listseg}(x, y)$ would not be precise: $\text{listseg}(x, x)$ could be true of a heap containing a non-empty circular list from x to x (and nothing else), and also of the empty heap, a proper subheap. For this reason, the list segment predicate in [36] is not precise, if we wrap it in an existential quantifier over the sequence parameter.

If p is a precise predicate then there can be at most one way to split any given heap up in such a way as to satisfy $p * q$; the splitting, if there is one, must give p the unique subheap satisfying it. This leads to an important property of precise predicates.

LEMMA 3. *A predicate p is precise if and only if $p * -$ distributes over \wedge :*

$$\text{for all predicates } q \text{ and } r, \text{ we have } p * (q \wedge r) = (p * q) \wedge (p * r).$$

We also have closure properties of precise predicates.

LEMMA 4. *For all precise predicates p and q , all (possibly imprecise) predicates r , and boolean expressions B , all the predicates $p \wedge r$, $p * q$, and $(B \wedge p) \vee (\neg B \wedge q)$ are precise.*

10 Soundness

All of the proof rules from Section 3.2 are sound in the denotational model. The main result of the paper concerns the hypothetical frame rule and, by implication, the modular procedure rule.

THEOREM 5.

- (a) *The hypothetical frame rule is sound for fixed preconditions p_1, \dots, p_n if and only if p_1, \dots, p_n are all precise.*
- (b) *The hypothetical frame rule is sound for a fixed invariant r if and only if r is precise.*

Theorem 5(a) addresses point 1 from Section 6: it rules out the precondition $0 \vee 1$ in the conundrum, which is not precise. Theorem 5(b) addresses point 2: it rules out the invariant true , which is not precise. And this result covers the queue and memory manager examples, where the preconditions and invariants are all precise.

There are two main concepts used in the proof of the theorem.

- **The greatest relation.** We identify the greatest relation for a specification $\{p\}k\{q\}[X]$, which is the largest local relation satisfying it. This allows us to reduce the truth of a sequent, which officially involves quantification over all environments, to the truth of a single triple for a single environment.
- **Simulation.** To show the soundness of the hypothetical frame rule we need to connect the meaning of a command in one context to its meaning in another with an additional invariant and additional modifies sets. We develop a notion of simulation relation between commands to describe this connection.

In the next two subsections we define these concepts and state some of their properties, and then we sketch their relevance in the proof of the theorem.

10.1 The Greatest Relation

For each specification $\{p\} - \{q\}[X]$, define $\text{great}(p, q, X)$

$$\begin{aligned} & (s, h)[\text{great}(p, q, X)]\text{fault} \\ \xrightleftharpoons[\text{def}]{\quad} & (s, h) \not\models p * \text{true} \\ & (s, h)[\text{great}(p, q, X)](s', h') \\ \xrightleftharpoons[\text{def}]{\quad} & (1) s(y) = s'(y) \text{ for all variables } y \notin X; \text{ and} \\ & (2) \forall h_p, h_1. (h_p * h_1 = h \wedge (s, h_p) \in p) \\ & \implies \exists h'_q. h'_q \# h_1 \wedge h'_q * h_1 = h' \wedge (s', h'_q) \in q \end{aligned}$$

The first equivalence says that $\text{great}(p, q, X)$ is safe at (s, h) just when p holds in (s, h_p) for some subheap h_p of h . The second equivalence is about state changes. The condition (1) means that $\text{great}(p, q, X)$ can modify only those variables in X . Condition (2) says that $\text{great}(p, q, X)$ demonically chooses a subheap h_p of the initial heap h that satisfies p (i.e., $(s, h_p) \in p$), and disposes all cells in h_p ; then, it angelically picks from q a new heap h'_q (i.e., $(s', h'_q) \in q$) and allocates h'_q to get the final heap h' .

LEMMA 6. *The relation $\text{great}(p, q, X)$ is in LRel, and satisfies $\{p\} - \{q\}$ and $\text{modifies}(-, X)$. Moreover, it is the greatest such: for all local relations c in LRel, we have that $\{p\}c\{q\} \wedge \text{modifies}(c, X) \implies c \subseteq \text{great}(p, q, X)$.*

The *greatest* environment for a context Γ is the largest environment, in the pointwise order, satisfying all the procedure specifications in Γ . It maps k to $\text{great}(p, q, X)$ when $\{p\}k\{q\}[X] \in \Gamma$; otherwise, it maps k to the top relation $\text{States} \times (\text{States} \cup \{\text{fault}\})$. Greatest environments give us a simpler way to interpret sequents and proof rules. A sequent $\Gamma \vdash \{p\}C\{q\}$ holds just if the triple $\{p\}(\llbracket C \rrbracket \eta)\{q\}$ holds for the greatest environment η satisfying Γ , leading to

PROPOSITION 7. *For all predicates p, q, p' and q' , commands C , and contexts Γ and Γ' , we have the following equivalence: the proof rule*

$$\frac{\Gamma \vdash \{p\}C\{q\}}{\Gamma' \vdash \{p'\}C\{q'\}}$$

holds if and only if we have $\{p\}[\llbracket C \rrbracket \eta]\{q\} \implies \{p'\}[\llbracket C \rrbracket \eta']\{q'\}$ for the greatest environments η and η' that, respectively, satisfy Γ and Γ' .

10.2 Simulation

Let $R : \text{States} \leftrightarrow \text{States}$ be a binary relation between states. For c, c_1 in LRel, we say that c_1 *simulates* c upto R , denoted $c[\text{sim}(R)]c_1$, just if the following properties hold:

- **Generalized Safety Monotonicity:** if $(s, h)[R](s_1, h_1)$, and c is safe at (s, h) , then c_1 is safe at (s_1, h_1) .
- **Generalized Frame Property:** if $(s, h)[R](s_1, h_1)$, c is safe at (s, h) , and $(s_1, h_1)[c_1](s'_1, h'_1)$, then there is a state (s', h') such that $(s, h)[c](s', h')$ and $(s', h')[R](s'_1, h'_1)$.

$c[\text{sim}(R)]c_1$ says that for R -related initial states (s, h) and (s_1, h_1) , when we have enough resources at (s, h) to run c safely, we also have enough resources at (s_1, h_1) to run c_1 safely; and in that case, every state transition from (s_1, h_1) in c_1 can be tracked by a transition from (s, h) in c .

Suppose r is a predicate. The following relation R_r plays a central role in the analysis of the hypothetical frame rule.

$$(s, h)[R_r](s_1, h_1) \stackrel{\text{def}}{\iff} s = s_1 \wedge \exists h_r. h_1 = h * h_r \wedge (s, h_r) \in r$$

The next result gives us a way to connect hypotheses in the premise and conclusion of the hypothetical rule, and additionally provides the characterization of precise predicates that is at the core of Theorem 5.

PROPOSITION 8.

- (a) *A predicate p is precise if and only if*

$$\text{great}(p, q, X)[\text{sim}(R_r)]\text{great}(p * r, q * r, X)$$

holds for for all predicates r and q , and sets X of variables.

- (b) *A predicate r is precise if and only if*

$$\text{great}(p, q, X)[\text{sim}(R_r)]\text{great}(p * r, q * r, X)$$

holds for all predicates p, q and sets X of variables.

The detailed proof of Theorem 5 relies on developing machinery that allows us to apply this key proposition; this development, and the proof of the proposition itself, is nontrivial, and will be left to the full paper. However, the relevance of the proposition can be seen by considering a special case of the hypothetical frame rule, for the key case of procedure call, and where the modified variables are held fixed:

$$\frac{\{p_1\}k\{q_1\}[X_1] \vdash \{p\}k\{q\}}{\{p_1 * r\}k\{q_1 * r\}[X_1] \vdash \{p * r\}k\{q * r\}}$$

We give a proof of the following proposition about this special case; it is the central step for showing Theorem 5.

PROPOSITION 9.

- (a) *The above special case of the hypothetical frame rule is sound for a fixed precondition p_1 if and only if p_1 is precise.*
- (b) *The above special case is sound for a fixed invariant r if and only if r is precise.*

Note that the if direction of the proposition is implied by the same direction of Theorem 5, and that for the only-if direction, the proposition implies the theorem. The proof of this proposition uses a lemma that characterizes $\text{sim}(R_r)$ using Hoare triples.

LEMMA 10. *Local relations c and c_1 are related by $\text{sim}(R_r)$ if and only if for all predicates p, q , we have*

$$\{p\}c\{q\} \implies \{p * r\}c_1\{q * r\}.$$

Proof: [of Proposition 9]

Using Proposition 7 and Lemma 10, we can simplify the above special case of the hypothetical frame rule as follows:

$$\begin{aligned}
 & \text{for all predicates } p, q, \\
 & \quad \text{if } \{p_1\}k_1\{q_1\}[X_1] \vdash \{p\}k\{q\} \text{ holds,} \\
 & \quad \text{then } \{p_1 * r\}k_1\{q_1 * r\}[X_1] \vdash \{p * r\}k\{q * r\} \text{ holds} \\
 \iff & (\because \text{Proposition 7}) \\
 & \text{for all predicates } p, q, \\
 & \quad \text{if } \{p\}\text{great}(p_1, q_1, X_1)\{q\} \text{ holds,} \\
 & \quad \text{then } \{p * r\}\text{great}(p_1 * r, q_1 * r, X_1)\{q * r\} \text{ holds} \\
 \iff & (\because \text{Lemma 10}) \\
 & \text{great}(p_1, q_1, X_1)[\text{sim}(R_r)]\text{great}(p_1 * r, q_1 * r, X_1)
 \end{aligned}$$

Now, Proposition 8(a) gives (a) of this proposition, and Proposition 8(b) gives (b) of this proposition. ■

10.3 Supported and Intuitionistic Predicates

There is a relaxation of the notion of precise predicate that can be used to provide further sufficient conditions for soundness. A predicate is *supported* if, for any stack and heap, the collection of sub-heaps making it true (while holding the stack constant) is empty or has a least element. A predicate is *intuitionistic* if it is closed under heap extension.

THEOREM 11. *The hypothetical frame rule is sound in the following cases:*

- (a) *the preconditions p_1, \dots, p_n are supported, and the postconditions q_1, \dots, q_n are intuitionistic; or*
- (b) *the resource invariant r is supported, and the postconditions q_1, \dots, q_n are intuitionistic.*

Notice that the first point does not contradict the only if part of Theorem 5(a), because it mentions postconditions in addition to preconditions. Likewise, the second point does not contradict Theorem 5(b), because it mentions postconditions as well as the resource invariant. This result about supported predicates would give us a version of the hypothetical frame rule appropriate when we are not interested in nailing down definite portions of memory using assertions, as might be the case in a garbage-collected language.

11 Related and Future Work

As we have emphasized, reliance on fixed resource partitioning has been an obstacle to the development of modular methods of program specification that are applicable to widely used programming languages. Because the separating conjunction $*$ is a logical connective, which depends on the state, it allows us to describe situations where the partition between a module and its clients changes over time. For example, with a resource manager module the resources transfer back and forth between the module and a client, as allocation and deallocation operations are performed, but correct operating relies on separation being maintained at all times.

A different reaction to the limitations of fixed partitioning has been the development of the assume-guarantee method of reasoning about program components [24, 21]. While this has proven successful, we are unsure whether it could be profitably applied to mutable data structures with embedded pointers. In any case, when partitioning can be ensured, be it fixed or dynamic, an invariant-based methodology leads to pleasantly modular specifications and proofs.

Perhaps the most significant previous work that addresses information hiding in program logics, and that confronts mutable data structures, is that of Leino and Nelson [23] (also, [12]). They use abstract variables (like our use of the variable Q in Table 3) to specify modules, and they develop a subtle notion of “modular soundness” that identifies situations when clients cannot access the internal representation of a module. This much is similar in spirit to what we are attempting, but on the technical level we are not at all sure if there is any relationship between the separating conjunction and their notion of modular soundness.

The information-hiding problems caused by pointers have been a concern for a number of years in the object-oriented types community (e.g., [19, 10, 15]). A focal point of that work has been a concept of “confinement”, which disallows or controls pointers into data representations. Some confinement systems use techniques similar to regions, with control over the number and direction of pointers across region boundaries.

The advantage of confinement schemes is their use of static typing, or static analysis, to provide algorithmic guarantees of information hiding properties. Conversely, separation logic is more flexible, not just because it is based on logic rather than types, but also because it allows any number of pointers from the outside, requiring only that these pointers not be dereferenced without permission. Current confinement schemes have difficulty with ownership transfer that involves aliasing (because they tend to rely on “unique” pointers), such as a program that disposes all of the elements in a graph, or with examples where resource partitioning depends on arithmetic properties.

Recent work on the semantics of confinement uses heap partitioning in an essential way [3, 33], thus suggesting the prospect of a deeper connection between type systems for confinement and logics of separation. There is also the possibility of promoting the heap partitioning operation $*$ used in the semantic models to a type operator in the source language; an immediate step could even be attempted to obtain a form of alias types with information hiding [37]. Further unification along these lines could be valuable.

It is striking that many proof systems for object-oriented languages work by exposing class invariants or other descriptions of internal states at method call sites (e.g., [13, 34]). Of course, the developers of such systems have rightly been careful, as unsoundness can very easily result if one incorrectly hides invariants. It seems plausible, however, that taking explicit account of separation or confinement could lead to an improved logic of objects.

Further afield in aims, but closer in technique, are logics of mobile and concurrent processes that have been developed by Cardelli, Caires and Gordon [9, 5]; related ideas have also been used to study semi-structured data [8, 7]. Cardelli et. al. use subsets of a commutative monoid, as in the general models of bunched logic [31, 32], but the interaction between logic and program dynamics is very different to that here. The models of [9, 5] do not satisfy the properties (such as the frame property) that drive our approach to information hiding. Furthermore, although the “pointers from outside” phenomenon certainly occurs in their setting, based as it is on the π -calculus, they do not use the conjunction $*$ (or $|$ in their notation) to control these pointers/names; rather, they employ a form of new name quantifier, following Gabbay and Pitts [14]. Despite the surface similarity of logical structure, we do not feel that we fully understand the relationship between the two approaches.

An intriguing question is if there is a link between the hypothet-

ical frame rule and the data abstraction provided by polymorphic types. Polymorphic typing can be used to hide the type in a data abstraction [35, 25], but this is not the same thing as hiding dynamic resources. For example, if we hide the type of a reference, polymorphic typing does not guarantee that the reference is not aliased, and accessible from outside the data abstraction. Still, the hypothetical frame rule ensures that a proof of client code can be used with any (precise) representation invariant, so there is an obvious intuitive analogy with polymorphic functions; the client should be parametrically polymorphic in the resource invariant. If this analogy can be made precise it may provide the basis for an approach to data refinement, and encapsulated components as first-class values, that accounts for mutable structures and dynamic ownership transfer.

We have stayed in a sequential setup in this paper, but the ideas seem relevant to concurrent programming. Indeed, the treatment by Hoare in [17] revolves around the concept of spatial separation, and the work here grew out of an attempt to directly adapt that approach, and its extension by Owicky and Gries [27], to separation logic. In unpublished notes from August 2001, O’Hearn described proof rules for concurrency using $*$ to express heap separation, and showed program proofs where storage moved from one process to another. The proof rules were not published, because O’Hearn was unable to establish their soundness. Then, in August 2002, Reynolds showed that the rules were unsound if used without restriction, and this lead to our focus on precise assertions. Both the promise and subtlety of the proof rules had as much to do with information hiding as concurrency, and it seemed unwise to attempt to tackle both at the same time. At the time of Reynolds’s discovery we had already begun work on the hypothetical frame rule, and the counterexample appears here as the conundrum in Section 6.

Work is underway on the semantics of the concurrent logic, and we are hopeful that a thorough account of the concurrency proof rules will be forthcoming.

Finally, in this paper we have concentrated on program proving, but there have been striking successes in software model checking [2, 11], which use abstraction to bridge the gap between infinite state language models and the finite state models expected by model checking algorithms; in essence, abstract interpretations are chosen, and sometimes refined, that allow for the synthesis of loop invariants. We wonder if one might synthesize resource invariants describing heap-sensitive properties as well, and use this to partition the model checking effort. For this to be workable the immediate challenge is to devise expressive heap abstractions [38] that are compatible with an information-hiding rule like the hypothetical frame rule.

Acknowledgements. We have benefitted greatly from discussions with Josh Berdine, Richard Bornat and Cristiano Calcagno. O’Hearn’s research was supported by the EPSRC project “Local Reasoning about State”. Reynolds’s research was partially supported by an EPSRC Visiting Fellowship at Queen Mary, University of London, by National Science Foundation Grant CCR-0204242, and by the Basic Research in Computer Science (<http://www.brics.dk/>) Centre of the Danish National Research Foundation. Yang was supported by grant No. R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation. Yang would like to thank EPSRC for supporting his visit to University of London in 2002, during which he first got involved in this work.

12 References

- [1] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *18th LICS*, 2003.
- [2] T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. *Proceedings of the Workshop on Advances in Verification*, 2000.
- [3] A. Banerjee and D. Naumann. Representation independence, confinement and access control. In *29th POPL*, 2002.
- [4] R. Bornat. Proving pointer programs in Hoare logic. *Mathematics of Program Construction*, 2000.
- [5] L. Cardelli and L. Caires. A spatial logic for concurrency. In *TACS’01*, LNCS 2255:1–37, Springer, 2001.
- [6] L. Cardelli, P. Gardner, and G. Ghelli. Querying trees with pointers. Unpublished notes, 2003.
- [7] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. *Proceedings of ICALP’02*.
- [8] L. Cardelli and G. Ghelli. A query language for semistructured data based on the ambient logic. *Proceedings of ESOP’01*.
- [9] L. Cardelli and A. D. Gordon. Anytime, anywhere. Modal logics for mobile ambients. In *27th POPL*, 2000.
- [10] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. *ECOOP*, LNCS 2072, 2001.
- [11] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
- [12] K.R.M. Leino D. Detlefs and G. Nelson. Wrestling with rep exposure. Digital SRC Research Report 156, 1998.
- [13] F. de Boer. A WP calculus for OO. In *FOSSACS*, 1999.
- [14] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In *14th LICS*, 1999.
- [15] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. *OOPSLA*, pp241–253, 2001.
- [16] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica* 4, 271–281, 1972.
- [17] C.A.R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques*. Academic Press, 1972.
- [18] C.A.R. Hoare. Monitors: An operating system structuring concept. *CACM*, 17(10):549–557, October 1974.
- [19] J. Hogg, D. Lea, R. Holt, A. Wills, and D. de Champeaux. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, April, 1992.
- [20] S. Isthiaq and P.W. O’Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, 2001.
- [21] C.B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.
- [22] B. Kernighan and D. Ritchie. *The C programming language*. Prentice Hall, 1988.
- [23] K.R.M. Leino and G. Nelson. Data abstraction and information hiding. *ACM TOPLAS* 24(5): 491–553, 2002.
- [24] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions of Software Engineering*, July, 1981.
- [25] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM TOPLAS* 10(3):470–502, 1988.
- [26] P. O’Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of Computer Science Logic*, pages 1–19, 2001. LNCS 2142.
- [27] S. Owicky and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
- [28] D.L. Parnas. Information distribution aspects of design methodology. *IFIP Congress (1) 1971*: 339–344, 1972.
- [29] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972.
- [30] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

- [31] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Kluwer Applied Logic series, 2002.
- [32] D.J. Pym, P.W. O’Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *TCS*, to appear, 2003.
- [33] U. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Proceedings of ESOP*, 2003.
- [34] B. Reus, M. Wirsing, and R. Hennicker. A Hoare calculus for verifying Java realizations of OCL-constrained design models. *FASE Proceedings*, LNCS 2029, pp300–316, 2001.
- [35] J.C. Reynolds. Types, abstraction and parametric polymorphism. *IFIP Proceedings*, pp513–523, 1983.
- [36] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. Invited Paper, *17th LICS*, 2002.
- [37] D. Walker and J.G. Morrisett. Alias types for recursive data structures. In *Types in Compilation*, pp177–206, 2000.
- [38] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Compiler Construction*, pp1–17, 2000.
- [39] H. Yang, and P. O’Hearn. A semantic basis for local reasoning. In *Proceedings of FOSSACS’02*, pp402–416, 2002.

13 Appendix: Variable Conditions

13.1 Side Conditions and Modifies Sets

We now clarify the side conditions for the hypothetical frame rule. To begin with, note that in the rule we are using comma between the X_i and Y for union of disjoint sets; the form of the rule therefore assumes that X_i and Y are disjoint.

The disjointness requirement for Y enforces that we do not observe the changes of a variable in Y while reasoning about C ; as a result, reasoning in client code is independent of variables in Y . We give a technical definition of several variants on a notion of disjointness of a set of variables X from a set of variables, a command, a predicate, or a context. X is disjoint from a set Y if their variables do not overlap; X is disjoint from a command C if X does not intersect with the free variables of C ; X is disjoint from predicate r if the predicate is invariant under changes to values of variables in X ; X is disjoint from context Γ if for all $\{p\}k\{q\}[Y]$ in Γ , X is disjoint from p , q and Y . This defines the second side condition.

The first side condition can be made rigorous with a relativized version of the usual notion of set of variables modified by a command. We describe this using a set $\text{Modifies}(C)(\Gamma; \Gamma')$ of variables associated with each command, where we split the context into two parts. The two most important clauses in the definition concern procedure call.

$$\begin{aligned}\text{Modifies}(k)(\Gamma; \Gamma') &= X, & \text{if } \{p\}k\{q\}[X] \in \Gamma \\ \text{Modifies}(k)(\Gamma; \Gamma') &= \{\}, & \text{if } \{p\}k\{q\}[X] \in \Gamma'\end{aligned}$$

The upshot is that $\text{Modifies}(C)(\Gamma; \Gamma')$ reports those variables modified by C , except that it doesn’t count any procedure calls for procedures in Γ' .

For the other commands, the relativized notion of modifies set is defined usual. For a compound command C with immediate subcommands C_1, \dots, C_n , the set $\text{Modifies}(C)(\Gamma; \Gamma')$ is the union $\cup_i \text{Modifies}(C_i)(\Gamma; \Gamma')$. Two of the basic commands are as follows:

$$\text{Modifies}(x := E)(\Gamma; \Gamma') = \{x\} \quad \text{Modifies}([x] := E)(\Gamma; \Gamma') = \{\}$$

For $[x] := E$ the modifies set is empty because the command alters the heap but not the stack.

We are now in a position to state the first side condition rigorously: it means

$\text{Modifies}(C)(\Gamma; \{p_1\}k_1\{q_1\}[X_1], \dots, \{p_n\}k_n\{q_n\}[X_n])$
is disjoint from r .

The modifies conditions for the ordinary frame and recursive procedure rules do not mention the “except through” clause. These can be formalized by taking Γ' to be empty in $\text{Modifies}(C)(\Gamma; \Gamma')$.

An important point is that the free variables of the resource invariant are allowed to overlap with the X_i . This often happens when using abstract variables to specify the behaviour of a module, as exemplified by the treatment of the abstract variable Q in the queue module in Table 3.

The complexity of modifies clauses is a general irritation in program logic, and one might feel that this problem with modifies clauses could be easily avoided, simply by doing away with assignment to variables, so that the heap component is the only part of the state that changes. While this is easy to do semantically, obtaining a satisfactory program logic is not as straightforward. The most important point is the treatment of abstract variables. For example, in the queue module the variable q is used in interface specifications as well as the invariant. If we were to try to place this variable into the heap then separation would not allow us to have it in both an interface specification and an invariant. If some other approach could be developed as an alternative to the changing abstract variables, that was itself not more complex, then perhaps we could finally do away with modifies conditions.

In situations where one wants to capture only “structural integrity” properties of data structures, rather than correctness properties, it is often possible to avoid abstract variables. For example, one sometimes wants to ensure, for example, that a data structure has the correct shape and has no dangling pointers, without giving a complete description of the data that is represented. Because abstract variables are not required (or less often required) in such situations we might get some way with a logic simpler than the one here, that does not require modifies clauses.

13.2 On Existentials and Free Variables

In [26, 36] there is an inference rule for introducing existential variables in preconditions and postconditions.

$$\frac{\{p\}C\{q\}}{\{\exists x.p\}C\{\exists x.q\}} \quad x \not\in \text{free}(C)$$

The side condition cannot be stated in the formalism of this paper. For, a procedure specification $\{p\}k\{q\}[X]$ identifies the variables, X , that k might modify, but not those that k might read from.

We can get around this problem by adding a free variable component to the sequent form, thus having

$$(Y) \Gamma \vdash \{p\}C\{q\}.$$

This constrains the variables appearing in C and all the procedures k_i , but not the preconditions and postconditions. This would allow us to describe the existential rule as

$$\frac{(Y) \Gamma \vdash \{p\}C\{q\}}{(Y) \Gamma \vdash \{\exists x.p\}C\{\exists x.q\}} \quad x \not\in Y$$

Another reasonable approach is to have a distinct class of “logical” variables, that cannot be assigned to in programs. For technical simplicity, we do not explicitly pursue either of these extensions in the current paper.

Separation Logic: A Logic for Shared Mutable Data Structures

John C. Reynolds*
Computer Science Department
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

In joint work with Peter O’Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure.

The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a “separating conjunction” that asserts that its subformulas hold for disjoint parts of the heap, and a closely related “separating implication”. Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing.

In this paper, we will survey the current development of this program logic, including extensions that permit unrestricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising future directions.

1. Introduction

The use of shared mutable data structures, i.e., of structures where an updatable field can be referenced from more than one point, is widespread in areas as diverse as systems programming and artificial intelligence. Approaches to reasoning about this technique have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size. (A partial bibliography is given in Reference [28].)

The problem faced by these approaches is that the correctness of a program that mutates data structures usually

*Portions of the author’s own research described in this survey were supported by National Science Foundation Grant CCR-9804014, and by the Basic Research in Computer Science (<http://www.brics.dk/>) Centre of the Danish National Research Foundation.

depends upon complex restrictions on the sharing in these structures. To illustrate this problem, and our approach to its solution, consider a simple example. The following program performs an in-place reversal of a list:

$$\begin{aligned} j := \mathbf{nil} ; \mathbf{while } i \neq \mathbf{nil} \mathbf{do} \\ (k := [i + 1] ; [i + 1] := j ; j := i ; i := k). \end{aligned}$$

(Here the notation $[e]$ denotes the contents of the storage at address e .)

The invariant of this program must state that i and j are lists representing two sequences α and β such that the reflection of the initial value α_0 can be obtained by concatenating the reflection of α onto β :

$$\exists \alpha, \beta. \text{list } \alpha \ i \wedge \text{list } \beta \ j \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta,$$

where the predicate $\text{list } \alpha \ i$ is defined by induction on the length of α :

$$\text{list } \epsilon \ i \stackrel{\text{def}}{=} i = \mathbf{nil} \quad \text{list}(a \cdot \alpha) \ i \stackrel{\text{def}}{=} \exists j. i \hookrightarrow a, j \wedge \text{list } \alpha \ j$$

(and \hookrightarrow can be read as “points to”).

Unfortunately, however, this is not enough, since the program will malfunction if there is any sharing between the lists i and j . To prohibit this we must extend the invariant to assert that only \mathbf{nil} is reachable from both i and j :

$$\begin{aligned} (\exists \alpha, \beta. \text{list } \alpha \ i \wedge \text{list } \beta \ j \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta) \\ \wedge (\forall k. \mathbf{reach}(i, k) \wedge \mathbf{reach}(j, k) \Rightarrow k = \mathbf{nil}), \end{aligned} \tag{1}$$

where

$$\begin{aligned} \mathbf{reach}(i, j) &\stackrel{\text{def}}{=} \exists n \geq 0. \mathbf{reach}_n(i, j) \\ \mathbf{reach}_0(i, j) &\stackrel{\text{def}}{=} i = j \\ \mathbf{reach}_{n+1}(i, j) &\stackrel{\text{def}}{=} \exists a, k. i \hookrightarrow a, k \wedge \mathbf{reach}_n(k, j). \end{aligned}$$

Even worse, suppose there is some other list x , representing a sequence γ , that is not supposed to be affected by

the execution of our program. Then it must not share with either i or j , so that the invariant becomes

$$\begin{aligned} & (\exists \alpha, \beta. \text{list } \alpha i \wedge \text{list } \beta j \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta) \wedge \text{list } \gamma x \\ & \wedge (\forall k. \text{reach}(i, k) \wedge \text{reach}(j, k) \Rightarrow k = \text{nil}) \\ & \wedge (\forall k. \text{reach}(x, k) \wedge (\text{reach}(i, k) \vee \text{reach}(j, k)) \\ & \qquad \qquad \qquad \Rightarrow k = \text{nil}). \end{aligned} \quad (2)$$

Even in this trivial situation, where all sharing is prohibited, it is evident that this form of reasoning scales poorly.

The key to avoiding this difficulty is to introduce a novel logical operation $P * Q$, called *separating conjunction* (or sometimes *independent* or *spatial* conjunction), that asserts that P and Q hold for *disjoint* portions of the addressable storage. In effect, the prohibition of sharing is built into this operation, so that Invariant (1) can be written as

$$(\exists \alpha, \beta. \text{list } \alpha i * \text{list } \beta j) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta, \quad (3)$$

and Invariant (2) as

$$(\exists \alpha, \beta. \text{list } \alpha i * \text{list } \beta j * \text{list } \gamma x) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta. \quad (4)$$

In fact, one can go further: Using an inference rule called the “frame rule”, one can infer directly that the program does not affect the list x from the fact that assertions such as (3) do not refer to this list.

The central concept of a separating conjunction is implicit in Burstall’s early idea of a “distinct nonrepeating tree system” [2]. In lectures in the fall of 1999, I described the concept explicitly, and embedded it in a flawed extension of Hoare logic [16, 17]. Soon thereafter, an intuitionistic logic based on this idea was discovered independently by Ishtiaq and O’Hearn [19] and by myself [28]. Realizing that this logic was an instance of the logic of bunched implications [23, 26], Ishtiaq and O’Hearn also introduced a *separating implication* $P \multimap Q$.

The intuitionistic character of this logic implied a monotonicity property: that an assertion true for some portion of the addressable storage would remain true for any extension of that portion, such as might be created by later storage allocation.

In their paper, however, Ishtiaq and O’Hearn also presented a classical version of the logic that does not impose this monotonicity property, and can therefore be used to reason about explicit storage deallocation; they showed that this version is more expressive than the intuitionistic logic, since the latter can be translated into the classical logic.

In both the intuitionistic and classical version of the logic, addresses were assumed to be disjoint from integers, and to refer to entire records rather than particular fields, so that “address arithmetic” was precluded. More recently, I generalized the logic to permit reasoning about unrestricted address arithmetic, by regarding addresses as

integers which refer to individual fields [24, 30, 29]. It is this form of the logic that will be described and used in most of the present paper. We will also describe O’Hearn’s frame rule [24, 35, 34, 19], which permits local reasoning about components of programs.

Since these logics are based on the idea that the structure of an assertion can describe the separation of storage into disjoint components, we have come to use the term *separation logics*, both for the extension of predicate calculus with the separation operators and the resulting extension of Hoare logic. A more precise name might be *storage separation logics*, since it is becoming apparent that the underlying idea can be generalized to describe the separation of other kinds of resources [3, 11, 12, 9, 10].

2. The Programming Language

The programming language we will use is the simple imperative language originally axiomatized by Hoare [16, 17], extended with new commands for the manipulation of mutable shared data structures:

$$\begin{aligned} \langle \text{comm} \rangle ::= & \dots \\ | \langle \text{var} \rangle := \text{cons}(\langle \text{exp} \rangle, \dots, \langle \text{exp} \rangle) & \text{allocation} \\ | \langle \text{var} \rangle := [\langle \text{exp} \rangle] & \text{lookup} \\ | [\langle \text{exp} \rangle] := \langle \text{exp} \rangle & \text{mutation} \\ | \text{dispose } \langle \text{exp} \rangle & \text{deallocation} \end{aligned}$$

Semantically, we extend computational states to contain two components: a store (or stack), mapping variables into values (as in the semantics of the unextended simple imperative language), and a heap, mapping addresses into values (and representing the mutable structures).

In the early versions of separation logic, integers, atoms, and addresses were regarded as distinct kinds of value, and heaps were mappings from finite sets of addresses to nonempty tuples of values:

$$\begin{aligned} \text{Values} &= \text{Integers} \cup \text{Atoms} \cup \text{Addresses} \\ \text{where Integers, Atoms, and Addresses are disjoint} \\ \text{Heaps} &= \bigcup_{\substack{\text{fin} \\ A \subseteq \text{Addresses}}} (A \rightarrow \text{Values}^+). \end{aligned}$$

To permit unrestricted address arithmetic, however, in the version of the logic used in most of this paper we will assume that all values are integers, an infinite number of which are addresses; we also assume that atoms are integers that are not addresses, and that heaps map addresses

into single values:

$$\text{Values} = \text{Integers}$$

$$\text{Atoms} \cup \text{Addresses} \subseteq \text{Integers}$$

where Atoms and Addresses are disjoint

$$\text{Heaps} = \bigcup_{\substack{\text{fin} \\ A \subseteq \text{Addresses}}} (A \rightarrow \text{Values}).$$

(To permit unlimited allocation of records of arbitrary size, we require that, for all $n \geq 0$, the set of addresses must contain infinitely many consecutive sequences of length n . For instance, this will occur if only a finite number of positive integers are not addresses.) In both versions of the logic, we assume

$$\text{nil} \in \text{Atoms}$$

$$\text{Stores}_V = V \rightarrow \text{Values}$$

$$\text{States}_V = \text{Stores}_V \times \text{Heaps},$$

where V is a finite set of variables.

Our intent is to capture the low-level character of machine language. One can think of the store as describing the contents of registers, and the heap as describing the contents of an addressable memory. This view is enhanced by assuming that each address is equipped with an ‘‘activity bit’’; then the domain of the heap is the finite set of *active* addresses.

The semantics of ordinary and boolean expressions is the same as in the simple imperative language:

$$[e \in \langle \text{exp} \rangle]_{\text{exp}} \in (\bigcup_{V \supseteq \text{FV}(e)} \text{Stores}_V) \rightarrow \text{Values}$$

$$[[b \in \langle \text{boolexp} \rangle]]_{\text{bexp}} \in (\bigcup_{V \supseteq \text{FV}(b)} \text{Stores}_V) \rightarrow \{\text{true}, \text{false}\}$$

(where $\text{FV}(p)$ is the set of variables occurring free in the phrase p). In particular, expressions do not depend upon the heap, so that they are always well-defined and never cause side-effects.

Thus expressions do not contain notations, such as **cons** or $[-]$, that refer to the heap. It follows that none of the new heap-manipulating commands are instances of the simple assignment command $\langle \text{var} \rangle := \langle \text{exp} \rangle$ (even though we write them with the familiar operator $:=$). In fact, they will not obey Hoare’s inference rule for assignment. However, since they alter the store at the variable v , we will say that the commands $v := \text{cons}(\dots)$ and $v := [e]$, as well as $v := e$ (but not $[v] := e$ or **dispose** v) *modify* v .

A simple way to define the meaning of the new commands is by small-step operational semantics, i.e., by defining a transition relation \rightsquigarrow between *configurations*, which are either

- *nonterminal*: A command-state pair $\langle c, (s, h) \rangle$, where $\text{FV}(c) \subseteq \text{dom } s$.

- *terminal*: A state (s, h) or **abort**.

We write $\gamma \rightsquigarrow^* \gamma'$ to indicate that there is a finite sequence of transitions from γ to γ' , and $\gamma \uparrow$ to indicate that there is an infinite sequence of transitions beginning with γ .

In this semantics, the heap-manipulating commands are specified by the following inference rules:

- Allocation

$$\langle v := \text{cons}(e_1, \dots, e_n), (s, h) \rangle$$

$$\rightsquigarrow ([s \mid v: \ell], [h \mid \ell: [[e_1]]_{\text{exp}} s \mid \dots \mid \ell+n-1: [[e_n]]_{\text{exp}} s]),$$

where $\ell, \dots, \ell+n-1 \in \text{Addresses} - \text{dom } h$.

- Lookup

$$\text{When } [[e]]_{\text{exp}} s \in \text{dom } h:$$

$$\langle v := [e], (s, h) \rangle \rightsquigarrow ([s \mid v: h([e]]_{\text{exp}} s)], h),$$

$$\text{When } [[e]]_{\text{exp}} s \notin \text{dom } h:$$

$$\langle v := [e], (s, h) \rangle \rightsquigarrow \text{abort}.$$

- Mutation

$$\text{When } [[e]]_{\text{exp}} s \in \text{dom } h:$$

$$\langle [e] := e', (s, h) \rangle \rightsquigarrow (s, [h \mid [[e]]_{\text{exp}} s: [[e']]_{\text{exp}} s]),$$

$$\text{When } [[e]]_{\text{exp}} s \notin \text{dom } h:$$

$$\langle [e] := e', (s, h) \rangle \rightsquigarrow \text{abort}.$$

- Deallocation

$$\text{When } [[e]]_{\text{exp}} s \in \text{dom } h:$$

$$\langle \text{dispose } e, (s, h) \rangle \rightsquigarrow (s, h \restriction (\text{dom } h - \{[[e]]_{\text{exp}} s\})),$$

$$\text{When } [[e]]_{\text{exp}} s \notin \text{dom } h:$$

$$\langle \text{dispose } e, (s, h) \rangle \rightsquigarrow \text{abort}.$$

(Here $[f \mid x: a]$ denotes the function that maps x into a and all other arguments y in the domain of f into $f y$. The notation $f \restriction S$ denotes the restriction of the function f to the domain S .)

The allocation operation activates and initializes n cells in the heap. Notice that, aside from the requirement that the addresses of these cells be consecutive and previously inactive, the choice of addresses is indeterminate.

The remaining operations all cause memory faults (denoted by the terminal configuration **abort**) if an inactive address is dereferenced or deallocated.

An important property of this language is the effect of restricting the heap on the execution of a command. Essentially, if the restriction removes an address that is dereferenced or deallocated by the command, then the restricted execution aborts; otherwise, however, the executions are similar, except for the presence of unchanging extra heap cells in the unrestricted execution.

To state this property precisely, we write $h_0 \perp h_1$ to indicate that the heaps h_0 and h_1 have disjoint domains, and $h_0 \cdot h_1$ to indicate the union of such heaps. Then, when $h_0 \subseteq h$,

- If $\langle c, (s, h) \rangle \rightsquigarrow^* \text{abort}$, then $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$.
- If $\langle c, (s, h) \rangle \rightsquigarrow^* (s', h')$ then $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$ or $\langle c, (s, h_0) \rangle \rightsquigarrow^* (s', h'_0)$, where $h'_0 \perp h_1$ and $h' = h'_0 \cdot h_1$.
- If $\langle c, (s, h) \rangle \uparrow$ then either $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$ or $\langle c, (s, h_0) \rangle \uparrow$.

3. Assertions and their Inference Rules

In addition to the usual formulae of predicate calculus, including boolean expressions and quantifiers, we introduce four new forms of assertion that describe the heap:

$\langle \text{assert} \rangle ::= \dots$	
emp	empty heap
$\langle \text{exp} \rangle \mapsto \langle \text{exp} \rangle$	singleton heap
$\langle \text{assert} \rangle * \langle \text{assert} \rangle$	separating conjunction
$\langle \text{assert} \rangle -* \langle \text{assert} \rangle$	separating implication

Because of these new forms, the meaning of an assertion (unlike that of a boolean expression) depends upon both the store and the heap:

$$\llbracket p \in \langle \text{assert} \rangle \rrbracket_{\text{asrt}} \in (\bigcup_{\substack{V \in \text{fin} \\ V \supseteq \text{FV}(p)}} \text{Stores}_V) \rightarrow \text{Heaps} \rightarrow \{\text{true}, \text{false}\}.$$

Specifically, **emp** asserts that the heap is empty:

$$\llbracket \text{emp} \rrbracket_{\text{asrt}} s h \text{ iff } \text{dom } h = \{\},$$

$e \mapsto e'$ asserts that the heap contains one cell, at address e with contents e' :

$$\llbracket e \mapsto e' \rrbracket_{\text{asrt}} s h \text{ iff }$$

$$\text{dom } h = \{\llbracket e \rrbracket_{\text{exp}} s\} \text{ and } h(\llbracket e \rrbracket_{\text{exp}} s) = \llbracket e' \rrbracket_{\text{exp}} s,$$

$p_0 * p_1$ asserts that the heap can be split into two disjoint parts in which p_0 and p_1 hold respectively:

$$\llbracket p_0 * p_1 \rrbracket_{\text{asrt}} s h \text{ iff }$$

$$\exists h_0, h_1. h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and }$$

$$\llbracket p_0 \rrbracket_{\text{asrt}} s h_0 \text{ and } \llbracket p_1 \rrbracket_{\text{asrt}} s h_1,$$

and $p_0 -* p_1$ asserts that, if the current heap is extended with a disjoint part in which p_0 holds, then p_1 will hold in the extended heap:

$$\llbracket p_0 -* p_1 \rrbracket_{\text{asrt}} s h \text{ iff }$$

$$\forall h'. (h' \perp h \text{ and } \llbracket p_0 \rrbracket_{\text{asrt}} s h') \text{ implies }$$

$$\llbracket p_1 \rrbracket_{\text{asrt}} s (h \cdot h').$$

When this semantics is coupled with the usual interpretation of the classical connectives, the result is an instance of the “resource semantics” of bunched implications as advanced by David Pym [23, 26].

It is useful to introduce several more complex forms as abbreviations:

$$e \mapsto - \stackrel{\text{def}}{=} \exists x'. e \mapsto x' \text{ where } x' \text{ not free in } e$$

$$e \hookleftarrow e' \stackrel{\text{def}}{=} e \mapsto e' * \text{true}$$

$$e \mapsto e_1, \dots, e_n$$

$$\stackrel{\text{def}}{=} e \mapsto e_1 * \dots * e + n - 1 \mapsto e_n$$

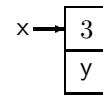
$$e \hookleftarrow e_1, \dots, e_n$$

$$\stackrel{\text{def}}{=} e \hookleftarrow e_1 * \dots * e + n - 1 \hookleftarrow e_n$$

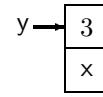
$$\text{iff } e \mapsto e_1, \dots, e_n * \text{true}.$$

By using \mapsto , \hookleftarrow , and the two forms of conjunction, it is easy to describe simple sharing patterns concisely:

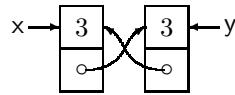
1. $x \mapsto 3, y$ asserts that x points to an adjacent pair of cells containing 3 and y (i.e., the store maps x and y into some values α and β , α is a address, and the heap maps α into 3 and $\alpha + 1$ into β):



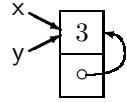
2. $y \mapsto 3, x$ asserts that y points to an adjacent pair of cells containing 3 and x :



3. $x \mapsto 3, y * y \mapsto 3, x$ asserts that situations (1) and (2) hold for separate parts of the heap:



4. $x \mapsto 3, y \mapsto 3, x$ asserts that situations (1) and (2) hold for the same heap, which can only happen if the values of x and y are the same:



5. $x \mapsto 3, y \mapsto 3, x$ asserts that either (3) or (4) may hold, and that the heap may contain additional cells.

There are also simple examples that reveal the occasionally surprising behavior of separating conjunction. Suppose s_x and s_y are distinct addresses, so that

$$h_1 = \{s_x, 1\} \quad \text{and} \quad h_2 = \{s_y, 2\}$$

are heaps with disjoint singleton domains. Then

If p is:	then $\llbracket p \rrbracket_{\text{asrt}} s h$ is:
$x \mapsto 1$	$h = h_1$
$y \mapsto 2$	$h = h_2$
$x \mapsto 1 * y \mapsto 2$	$h = h_1 \cdot h_2$
$x \mapsto 1 * x \mapsto 1$	false
$x \mapsto 1 \vee y \mapsto 2$	$h = h_1 \text{ or } h = h_2$
$x \mapsto 1 * (x \mapsto 1 \vee y \mapsto 2)$	$h = h_1 \cdot h_2$
$(x \mapsto 1 \vee y \mapsto 2) * (x \mapsto 1 \vee y \mapsto 2)$	$h = h_1 \cdot h_2$
$x \mapsto 1 * y \mapsto 2 * (x \mapsto 1 \vee y \mapsto 2)$	false
$x \mapsto 1 * \text{true}$	$h_1 \subseteq h$
$x \mapsto 1 * \neg x \mapsto 1$	$h_1 \subseteq h$.

To illustrate separating implication, suppose that an assertion p holds for a store that maps the variable x into an address α and a heap h that maps α into 16. Then

$$(x \mapsto 16) \multimap p$$

holds for the same store and the heap $h](\text{dom } h - \{\alpha\})$ obtained by removing α from the domain of h , and

$$x \mapsto 9 * ((x \mapsto 16) \multimap p)$$

holds for the same store and the heap $[h | \alpha: 9]$ that differs from h by mapping α into 9. (Thus, anticipating the concept of partial-correctness specification introduced in the next section, $\{x \mapsto 9 * ((x \mapsto 16) \multimap p)\} [x] := 16 \{p\}$.)

The inference rules for predicate calculus remain sound in this enriched setting. Before presenting sound rules for the new forms of assertions, however, we note two rules that fail. Taking p to be $x \mapsto 1$ and q to be $y \mapsto 2$, one can see

that neither contraction, $p \Rightarrow p * p$, nor weakening, $p * q \Rightarrow p$, are sound. Thus separation logic is a substructural logic. (It is not, however, a species of linear logic, since in linear logic $P \Rightarrow Q$ can be written $(!P) \multimap Q$, so the rule of dereliction gives the validity of $(P \multimap Q) \Rightarrow (P \Rightarrow Q)$. But in a state where $x \mapsto 1$ is true, $(x \mapsto 1) \multimap \text{false}$ is true but $(x \mapsto 1) \Rightarrow \text{false}$ is false [19].)

Sound axiom schemata for separating conjunction include commutative and associative laws, the fact that **emp** is a neutral element, and various distributive and semidistributive laws:

$$\begin{aligned} p_1 * p_2 &\Leftrightarrow p_2 * p_1 \\ (p_1 * p_2) * p_3 &\Leftrightarrow p_1 * (p_2 * p_3) \end{aligned}$$

$$\begin{aligned} p * \mathbf{emp} &\Leftrightarrow p \\ (p_1 \vee p_2) * q &\Leftrightarrow (p_1 * q) \vee (p_2 * q) \\ (p_1 \wedge p_2) * q &\Rightarrow (p_1 * q) \wedge (p_2 * q) \\ (\exists x. p) * q &\Leftrightarrow \exists x. (p * q) \text{ when } x \text{ not free in } q \\ (\forall x. p) * q &\Rightarrow \forall x. (p * q) \text{ when } x \text{ not free in } q. \end{aligned}$$

There is also an inference rule showing that separating conjunction is monotone with respect to implication:

$$\frac{p_1 \Rightarrow p_2 \quad q_1 \Rightarrow q_2}{p_1 * q_1 \Rightarrow p_2 * q_2},$$

and two further rules capturing the adjunctive relationship between separating conjunction and separating implication:

$$\frac{p_1 * p_2 \Rightarrow p_3}{p_1 \Rightarrow (p_2 \multimap p_3)} \quad \frac{p_1 \Rightarrow (p_2 \multimap p_3)}{p_1 * p_2 \Rightarrow p_3}.$$

There are several semantically defined classes of assertions that have useful special properties, and contain an easily defined syntactic subclass.

An assertion is said to be *pure* if, for any store, it is independent of the heap. Syntactically, an assertion is pure if it does not contain **emp**, \mapsto , or \multimap . The following axiom schemata show that, when assertions are pure, the distinction between the two kinds of conjunctions, or of implications, collapses:

$$\begin{array}{ll} p_1 \wedge p_2 \Rightarrow p_1 * p_2 & \text{when } p_1 \text{ or } p_2 \text{ is pure} \\ p_1 * p_2 \Rightarrow p_1 \wedge p_2 & \text{when } p_1 \text{ and } p_2 \text{ are pure} \\ (p \wedge q) * r \Leftrightarrow p \wedge (q * r) & \text{when } p \text{ is pure} \\ (p \multimap p_2) \Rightarrow (p_1 \Rightarrow p_2) & \text{when } p_1 \text{ is pure} \\ (p_1 \Rightarrow p_2) \Rightarrow (p_1 \multimap p_2) & \text{when } p_1 \text{ and } p_2 \text{ are pure.} \end{array}$$

We say that an assertion p is *intuitionistic* iff, for all stores s and heaps h and h' :

$$h \subseteq h' \text{ and } \llbracket p \rrbracket_{\text{asrt}}(s, h) \text{ implies } \llbracket p \rrbracket_{\text{asrt}}(s, h').$$

One can show that $p * \text{true}$ is the strongest intuitionistic assertion weaker than p , and that $\text{true} \multimap p$ is the weakest intuitionistic assertion stronger than p . Syntactically, If p and q are intuitionistic assertions, r is any assertion, and e and e' are expressions, then the following are intuitionistic assertions:

$$\begin{array}{ll} \text{Any pure assertion} & e \hookrightarrow e' \\ r * \text{true} & \text{true} \multimap r \\ p \wedge q & p \vee q \\ \forall v. p & \exists v. p \\ p * q & p \multimap q. \end{array}$$

For intuitionistic assertions, we have

$$\begin{array}{ll} (p * \text{true}) \Rightarrow p & \text{when } p \text{ is intuitionistic} \\ p \Rightarrow (\text{true} \multimap p) & \text{when } p \text{ is intuitionistic.} \end{array}$$

It should also be noted that, if we define the operations

$$\begin{aligned} \stackrel{i}{\rightarrow} p &\stackrel{\text{def}}{=} \text{true} \multimap (\neg p) \\ p \stackrel{i}{\Rightarrow} q &\stackrel{\text{def}}{=} \text{true} \multimap (p \Rightarrow q) \\ p \stackrel{i}{\Leftrightarrow} q &\stackrel{\text{def}}{=} \text{true} \multimap (p \Leftrightarrow q), \end{aligned}$$

then the assertions built from pure assertions and $e \hookrightarrow e'$, using these operations and $\wedge, \vee, \forall, \exists, *, \text{ and } \multimap$ form the image of Ishtiaq and O'Hearn's modal translation from intuitionistic separation logic to the classical version [19].

Yang [33, 34] has singled out the class of *strictly exact* assertions; an assertion q is strictly exact iff, for all s, h , and h' , $\llbracket q \rrbracket_{\text{asrt}} sh$ and $\llbracket q \rrbracket_{\text{asrt}} sh'$ implies $h = h'$. Syntactically, assertions built from expressions using \hookrightarrow and $*$ are strictly exact. The utility of this concept is that

$$(q * \text{true}) \wedge p \Rightarrow q * (q \multimap p) \quad \text{when } q \text{ is strictly exact.}$$

Strictly exact assertions also belong to the broader class of *domain-exact* assertions; an assertion q is domain-exact iff, for all s, h , and h' , $\llbracket q \rrbracket_{\text{asrt}} sh$ and $\llbracket q \rrbracket_{\text{asrt}} sh'$ implies $\text{dom } h = \text{dom } h'$. Syntactically, assertions built from expressions using $\hookrightarrow, *, \text{ and quantifiers}$ are domain-exact. When q is such an assertion, the semidistributive laws given earlier become full distributive laws:

$$\begin{aligned} (p_1 * q) \wedge (p_2 * q) &\Rightarrow (p_1 \wedge p_2) * q \\ &\quad \text{when } q \text{ is domain-exact} \end{aligned}$$

$$\forall x. (p * q) \Rightarrow (\forall x. p) * q \quad \text{when } x \text{ not free in } q \text{ and } q \text{ is domain-exact.}$$

Finally, we give axiom schemata for the predicate \hookrightarrow .

(Regrettably, these are far from complete.)

$$\begin{aligned} e_1 \hookrightarrow e'_1 \wedge e_2 \hookrightarrow e'_2 &\Leftrightarrow e_1 \hookrightarrow e'_1 \wedge e_1 = e_2 \wedge e'_1 = e'_2 \\ e_1 \hookrightarrow e'_1 * e_2 \hookrightarrow e'_2 &\Rightarrow e_1 \neq e_2 \\ \text{emp} &\Leftrightarrow \forall x. \neg(x \hookrightarrow -). \end{aligned}$$

Both the assertion language developed in this section and the programming language developed in the previous section are limited by the fact that all variables range over the integers. Later in this paper we will go beyond this limitation in ways that are sufficiently obvious that we will not formalize their semantics, e.g., we will use variables denoting abstract data types, predicates, and assertions in the assertion language, and variables denoting procedures in the programming language. (We will not, however, consider assignment to such variables.)

4. Specifications and their Inference Rules

We use a notion of program specification that is similar to that of Hoare logic, with variants for both partial and total correctness:

$$\begin{array}{ll} \langle \text{spec} \rangle ::= \{ \langle \text{assert} \rangle \} \langle \text{comm} \rangle \{ \langle \text{assert} \rangle \} & \text{partial} \\ | [\langle \text{assert} \rangle] \langle \text{comm} \rangle [\langle \text{assert} \rangle] & \text{total} \end{array}$$

Let $V = \text{FV}(p) \cup \text{FV}(c) \cup \text{FV}(q)$. Then

$$\begin{aligned} \{p\} c \{q\} \text{ holds iff } \forall (s, h) \in \text{States}_V. \llbracket p \rrbracket_{\text{asrt}} s h \text{ implies} \\ \neg(c, (s, h) \rightsquigarrow^* \text{abort}) \\ \text{and } (\forall (s', h') \in \text{States}_V. \\ c, (s, h) \rightsquigarrow^* (s', h') \text{ implies } \llbracket q \rrbracket_{\text{asrt}} s' h'), \end{aligned}$$

and

$$\begin{aligned} [p] c [q] \text{ holds iff } \forall (s, h) \in \text{States}_V. \llbracket p \rrbracket_{\text{asrt}} s h \text{ implies} \\ \neg(c, (s, h) \rightsquigarrow^* \text{abort}) \\ \text{and } \neg(c, (s, h) \uparrow) \\ \text{and } (\forall (s', h') \in \text{States}_V. \\ c, (s, h) \rightsquigarrow^* (s', h') \text{ implies } \llbracket q \rrbracket_{\text{asrt}} s' h'). \end{aligned}$$

Notice that specifications are implicitly quantified over both stores and heaps, and also (since allocation is indeterminate) over all possible executions. Moreover, any execution giving a memory fault falsifies both partial and total specifications.

As O'Hearn [19] paraphrased Milner, “Well-specified programs don't go wrong.” As a consequence, during the execution of programs that have been proved to meet their specifications, it is unnecessary to check for memory faults, or even to equip heap cells with activity bits (assuming that

programs are only executed in initial states satisfying the relevant precondition).

Roughly speaking, the fact that specifications preclude memory faults acts in concert with the indeterminacy of allocation to prohibit violations of record boundaries. But sometimes the notion of record boundaries dissolves, as in the following valid specification of a program that tries to form a two-field record by gluing together two one-field records:

```
{x ↦ − * y ↦ −}
if y = x + 1 then skip else
  if x = y + 1 then x := y else
    (dispose x ; dispose y ; x := cons(1, 2))
{x ↦ −, −}.
```

In our new setting, the command-specific inference rules of Hoare logic remain sound, as do such structural rules as

- Consequence

$$\frac{p' \Rightarrow p \quad \{p\} c \{q\} \quad q \Rightarrow q'}{\{p'\} c \{q'\}}.$$

- Auxiliary Variable Elimination

$$\frac{\{p\} c \{q\}}{\{\exists v. p\} c \{\exists v. q\}},$$

where v is not free in c .

- Substitution

$$\frac{\{p\} c \{q\}}{(\{p\} c \{q\}) / v_1 \rightarrow e_1, \dots, v_n \rightarrow e_n},$$

where v_1, \dots, v_n are the variables occurring free in p , c , or q , and, if v_i is modified by c , then e_i is a variable that does not occur free in any other e_j .

(All of the inference rules presented in this section are the same for partial and total correctness.)

An exception is what is sometimes called the “rule of constancy” [27, Section 3.3.5; 28, Section 3.5]:

$$\frac{\{p\} c \{q\}}{\{p \wedge r\} c \{q \wedge r\}},$$

where no variable occurring free in r is modified by c . It has long been understood that this rule is vital for scalability, since it permits one to extend a “local” specification of c , involving only the variables actually used by that command, by adding arbitrary predicates about variables that are not modified by c and will therefore be preserved by its execution.

Unfortunately, however, the rule of constancy is not sound for separation logic. For example, the conclusion of the instance

$$\frac{\{x \mapsto -\} [x] := 4 \{x \mapsto 4\}}{\{x \mapsto - \wedge y \mapsto 3\} [x] := 4 \{x \mapsto 4 \wedge y \mapsto 3\}}$$

is not valid, since its precondition does not preclude the aliasing that will occur if $x = y$.

O’Hearn realized, however, that the ability to extend local specifications can be regained at a deeper level by using separating conjunction. In place of the rule of constancy, he proposed the *frame rule*:

- Frame Rule

$$\frac{\{p\} c \{q\}}{\{p * r\} c \{q * r\}},$$

where no variable occurring free in r is modified by c .

By using the frame rule, one can extend a local specification, involving only the variables and *parts of the heap* that are actually used by c (which O’Hearn calls the *footprint* of c), by adding arbitrary predicates about variables and parts of the heap that are not modified or mutated by c . Thus, the frame rule is the key to “local reasoning” about the heap:

To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged [24].

Every valid specification $\{p\} c \{q\}$ is “tight” in the sense that every cell in its footprint must either be allocated by c or asserted to be active by p ; “locality” is the opposite property that everything asserted to be active belongs to the footprint. The role of the frame rule is to infer from a local specification of a command the more global specification appropriate to the larger footprint of an enclosing command.

To see the soundness of the frame rule [35, 34], assume $\{p\} c \{q\}$, and $\llbracket p * r \rrbracket_{\text{asrt}} s h$. Then there are h_0 and h_1 such that $h_0 \perp h_1$, $h = h_0 \cdot h_1$, $\llbracket p \rrbracket s h_0$ and $\llbracket r \rrbracket s h_1$.

- Suppose $\langle c, (s, h) \rangle \rightsquigarrow^* \text{abort}$. Then, by the property described at the end of Section 2, we would have $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$, which contradicts $\{p\} c \{q\}$ and $\llbracket p \rrbracket s h_0$.
- Suppose $\langle c, (s, h) \rangle \rightsquigarrow^* (s', h')$. As in the previous case, $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$ would contradict $\{p\} c \{q\}$ and $\llbracket p \rrbracket s h_0$, so that, by the property at the end of Section 2, $\langle c, (s, h_0) \rangle \rightsquigarrow^* (s', h'_0)$, where $h'_0 \perp h_1$ and $h' = h'_0 \cdot h_1$. Then $\{p\} c \{q\}$ and $\llbracket p \rrbracket s h_0$ implies that $\llbracket q \rrbracket s' h'_0$. Moreover, since $\langle c, (s, h) \rangle \rightsquigarrow^* (s', h')$, the stores s and s' will give the same value

to the variables that are not modified by c . Then, since these include all the free variables of r , $\llbracket r \rrbracket s h_1$ implies that $\llbracket r \rrbracket s' h_1$. Thus $\llbracket q * r \rrbracket s' h'$.

Yang [35, 34] has also shown that the frame rule is complete in the following sense: Suppose that all we know about a command c is that some partial correctness specification $\{p\} c \{q\}$ is valid. Then, whenever the validity of $\{p'\} c \{q'\}$ is a semantic consequence of this knowledge, $\{p'\} c \{q'\}$ can be derived from $\{p\} c \{q\}$ by use of the frame rule and the rules of consequence, auxiliary variable elimination, and substitution.

Using the frame rule, one can move from local versions of inference rules for the primitive heap-manipulating commands to equivalent global versions. For mutation, for example, the obvious local rule

- Mutation (local)

$$\overline{\{e \mapsto -\} [e] := e' \{e \mapsto e'\}}$$

leads directly to

- Mutation (global)

$$\overline{\{(e \mapsto -) * r\} [e] := e' \{(e \mapsto e') * r\}}.$$

(One can rederive the local rule from the global one by taking r to be `emp`.) Moreover, by taking r in the global rule to be $(e \mapsto e') \multimap p$ and using the valid implication $q * (q \multimap p) \Rightarrow p$, one can derive a third rule for mutation that is suitable for backward reasoning [19], since one can substitute any assertion for the postcondition p :

- Mutation (backwards reasoning)

$$\overline{\{(e \mapsto -) * ((e \mapsto e') \multimap p)\} [e] := e' \{p\}}.$$

(One can rederive the global rule from the backward one by taking p to be $(e \mapsto e') * r$ and using the valid implication $(p * r) \Rightarrow (p * (q \multimap (q * r)))$.)

A similar development works for deallocation, except that the global form is itself suitable for backward reasoning:

- Deallocation (local)

$$\overline{\{e \mapsto -\} \mathbf{dispose} e \{\mathbf{emp}\}}.$$

- Deallocation (global, backwards reasoning)

$$\overline{\{(e \mapsto -) * r\} \mathbf{dispose} e \{r\}}.$$

In the same way, one can give equivalent local and global rules for “noninterfering” allocation commands that modify “fresh” variables. Here we abbreviate e_1, \dots, e_n by \bar{e} .

- Allocation (noninterfering, local)

$$\overline{\{\mathbf{emp}\} v := \mathbf{cons}(\bar{e}) \{v \mapsto \bar{e}\}},$$

where v is not free in \bar{e} .

- Allocation (noninterfering, global)

$$\overline{\{r\} v := \mathbf{cons}(\bar{e}) \{(v \mapsto \bar{e}) * r\}},$$

where v is not free in \bar{e} or r .

However, to avoid the restrictions on occurrences of the assigned variable, or to give a backward-reasoning rule, or rules for lookup, we must introduce and often quantify additional variables. For both allocation and lookup, we can give equivalent rules of the three kinds, but the relevant derivations are more complicated than before since one must use auxiliary variable elimination, properties of quantifiers, and other laws.

In these rules we indicate substitution by priming metavariables denoting expressions and assertions, e.g., we write e'_i for $e_i/v \rightarrow v'$ and r' for $r/v \rightarrow v'$. We also abbreviate e_1, \dots, e_n by \bar{e} and e'_1, \dots, e'_n by \bar{e}' .

- Allocation (local)

$$\overline{\{v = v' \wedge \mathbf{emp}\} v := \mathbf{cons}(\bar{e}) \{v \mapsto \bar{e}'\}},$$

where v' is distinct from v .

- Allocation (global)

$$\overline{\{r\} v := \mathbf{cons}(\bar{e}) \{\exists v'. (v \mapsto \bar{e}') * r'\}},$$

where v' is distinct from v , and is not free in \bar{e} or r .

- Allocation (backwards reasoning)

$$\overline{\{\forall v'. (v' \mapsto \bar{e}) \multimap p'\} v := \mathbf{cons}(\bar{e}) \{p\}},$$

where v' is distinct from v , and is not free in \bar{e} or p .

- Lookup (local)

$$\overline{\{v = v' \wedge (e \mapsto v'')\} v := [e] \{v = v'' \wedge (e' \mapsto v'')\}},$$

where v , v' , and v'' are distinct.

- Lookup (global)

$$\overline{\{\exists v''. (e \mapsto v'') * (r/v' \rightarrow v)\} v := [e] \{\exists v'. (e' \mapsto v) * (r/v'' \rightarrow v)\}},$$

where v , v' , and v'' are distinct, v' and v'' do not occur free in e , and v is not free in r .

- Lookup (backwards reasoning)

$$\frac{\{\exists v'. (e \mapsto v') * ((e \mapsto v') \rightarrow p')\} v := [e] \{p\},}{\{\exists v'. (e \mapsto v') \wedge p'\} v := [e] \{p\}},$$

where v' is not free in e , nor free in p unless it is v .

Finally, since $e \mapsto v'$ is strictly exact, it is easy to obtain an equivalent but more succinct rule for backward reasoning about lookup:

- Lookup (alternative backward reasoning)

$$\frac{\{\exists v'. (e \mapsto v') \wedge p'\} v := [e] \{p\},}{\{\exists v'. (e \mapsto v') \wedge p'\} v := [e] \{p\}},$$

where v' is not free in e , nor free in p unless it is v .

For all four new commands, the backward reasoning forms give complete weakest preconditions [19, 34].

As a simple illustration, the following is a detailed proof outline of a local specification of a command that uses allocation and mutation to construct a two-element cyclic structure containing relative addresses:

```
{emp}
x := cons(a, a) ;
{x ↪ a, a}
y := cons(b, b) ;
{(x ↪ a, a) * (y ↪ b, b)}
{(x ↪ a, -) * (y ↪ b, -)}
[x + 1] := y - x ;
{(x ↪ a, y - x) * (y ↪ b, -)}
[y + 1] := x - y ;
{(x ↪ a, y - x) * (y ↪ b, x - y)}
{∃o. (x ↪ a, o) * (x + o ↪ b, - o)}.
```

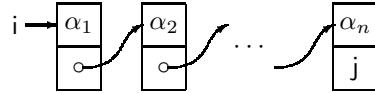
5. Lists

To specify a program adequately, it is usually necessary to describe more than the form of its structures or the sharing patterns between them; one must relate the states of the program to the abstract values that they denote. To do so, it is necessary to define the set of abstract values algebraically or recursively, and to define predicates on the abstract values by structural induction. Since these are standard methods of definition, we will treat them less formally than the novel aspects of our logic.

For lists, the relevant abstract values are sequences, for which we use the following notation: When α and β are sequences, we write

- ϵ for the empty sequence.
- $[x]$ for the single-element sequence containing x . (We will omit the brackets when x is not a sequence.)
- $\alpha \cdot \beta$ for the composition of α followed by β .
- α^\dagger for the reflection of α .
- $\#\alpha$ for the length of α .
- α_i for the i th component of α .

The simplest list structure for representing sequences is the *singly-linked* list. To describe this representation, we write list $\alpha(i, j)$ when there is a list segment from i to j representing the sequence α :



It is straightforward to define this predicate by induction on the structure of α :

$$\text{list } \epsilon(i, j) \stackrel{\text{def}}{=} \text{emp} \wedge i = j$$

$$\text{list } a \cdot \alpha(i, k) \stackrel{\text{def}}{=} \exists j. i \mapsto a, j * \text{list } \alpha(j, k),$$

and to prove some basic properties:

$$\text{list } a(i, j) \Leftrightarrow i \mapsto a, j$$

$$\text{list } \alpha \cdot \beta(i, k) \Leftrightarrow \exists j. \text{list } \alpha(i, j) * \text{list } \beta(j, k)$$

$$\text{list } \alpha \cdot b(i, k) \Leftrightarrow \exists j. \text{list } \alpha(i, j) * j \mapsto b, k.$$

(The second property is a composition law that can be proved by structural induction on α .)

In comparison with the definition of list in the introduction, we have generalized from lists to list segments, and we have used separating conjunction to prohibit cycles *within* the list segment. More precisely, when list $\alpha_1 \cdot \dots \cdot \alpha_n(i_0, i_n)$, we have

$$\exists i_1, \dots, i_{n-1}.$$

$$(i_0 \mapsto \alpha_1, i_1) * (i_1 \mapsto \alpha_2, i_2) * \dots * (i_{n-1} \mapsto \alpha_n, i_n).$$

Thus i_0, \dots, i_{n-1} are distinct, but i_n is not constrained, so that list $\alpha_1 \cdot \dots \cdot \alpha_n(i, i)$ may hold for any $n \geq 0$.

Thus the obvious properties

$$\text{list } \alpha(i, j) \Rightarrow (i = \text{nil} \Rightarrow (\alpha = \epsilon \wedge j = \text{nil}))$$

$$\text{list } \alpha(i, j) \Rightarrow (i \neq j \Rightarrow \alpha \neq \epsilon)$$

do not say whether α is empty when $i = j \neq \text{nil}$. However, there are some common situations that insure that α is empty when $i = j \neq \text{nil}$, for example, when j refers to a heap cell separate from the list segment, or to the beginning of a separate list:

$$\begin{aligned} \text{list } \alpha(i, j) * j \hookrightarrow - \Rightarrow (i = j \Leftrightarrow \alpha = \epsilon) \\ \text{list } \alpha(i, j) * \text{list } \beta(j, \text{nil}) \Rightarrow (i = j \Leftrightarrow \alpha = \epsilon). \end{aligned}$$

On the other hand, sometimes $i = j$ simply does not determine emptiness. For example, a cyclic buffer containing α in its active segment and β in its inactive segment is described by $\text{list } \alpha(i, j) * \text{list } \beta(j, i)$. Here, the buffer may be either empty or full when $i = j$.

The use of `list` is illustrated by a proof outline for a command that deletes the first element of a list:

```

{list a·α(i, k)}
{∃j. i ↦ a, j * list α(j, k)}
{i ↦ a * ∃j. i + 1 ↦ j * list α(j, k)}
j := [i + 1];
{i ↦ a * i + 1 ↦ j * list α(j, k)}
dispose i;
{i + 1 ↦ j * list α(j, k)}
dispose i + 1;
{list α(j, k)}
i := j
{list α(i, k)}

```

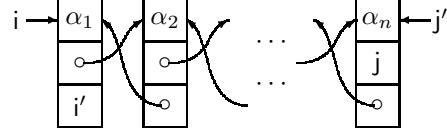
A more complex example is the body of the while command in the list-reversing program in the Introduction; here the final assertion is the invariant of the while command:

```

{∃α, β. (list α(i, nil) * list β(j, nil))
 ∧ α₀† = α† · β ∧ i ≠ nil}
{∃a, α, β. (list a·α(i, nil) * list β(j, nil))
 ∧ α₀† = (a·α)† · β}
{∃a, α, β, k. (i ↦ a, k * list α(k, nil) * list β(j, nil))
 ∧ α₀† = (a·α)† · β}
k := [i + 1];
{∃a, α, β. (i ↦ a, k * list α(k, nil) * list β(j, nil))
 ∧ α₀† = (a·α)† · β}
[i + 1] := j;
{∃a, α, β. (i ↦ a, j * list α(k, nil) * list β(j, nil))
 ∧ α₀† = (a·α)† · β}
{j := i; i := k}
{∃α, β. (list α(i, nil) * list β(j, nil)) ∧ α₀† = α† · β}.

```

A more elaborate representation of sequences is provided by *doubly-linked* lists. Here, we write $\text{dlist } \alpha(i, i', j, j')$ when α is represented by a doubly-linked list segment with a forward linkage (via second fields) from i to j , and a backward linkage (via third fields) from j' to i' :



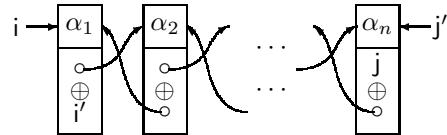
The inductive definition is

$$\begin{aligned} \text{dlist } \epsilon(i, i', j, j') &\stackrel{\text{def}}{=} \text{emp} \wedge i = j \wedge i' = j' \\ \text{dlist } a·\alpha(i, i', k, k') &\stackrel{\text{def}}{=} \exists j. i \mapsto a, j, i' * \text{dlist } \alpha(j, i, k, k'), \end{aligned}$$

from which one can prove the basic properties:

$$\begin{aligned} \text{dlist } a(i, i', j, j') &\Leftrightarrow i \mapsto a, j, i' \wedge i = j' \\ \text{dlist } \alpha·\beta(i, i', k, k') &\Leftrightarrow \\ &\quad \exists j, j'. \text{dlist } \alpha(i, i', j, j') * \text{dlist } \beta(j, j', k, k') \\ \text{dlist } \alpha·b(i, i', k, k') &\Leftrightarrow \\ &\quad \exists j'. \text{dlist } \alpha(i, i', k', j') * k' \mapsto b, k, j'. \end{aligned}$$

The utility of unrestricted address arithmetic is illustrated by a variation of the doubly-linked list, in which the second and third fields of each record are replaced by a single field giving the exclusive **or** of their contents. If we write $\text{xlist } \alpha(i, i', j, j')$ when α is represented by such an *xor-linked* list:



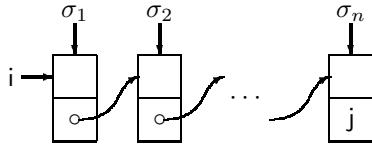
we can define this predicate by

$$\begin{aligned} \text{xlist } \epsilon(i, i', j, j') &\stackrel{\text{def}}{=} \text{emp} \wedge i = j \wedge i' = j' \\ \text{xlist } a·\alpha(i, i', k, k') &\stackrel{\text{def}}{=} \\ &\quad \exists j. i \mapsto a, (j \oplus i') * \text{xlist } \alpha(j, i, k, k'). \end{aligned}$$

The basic properties are analogous to those for `dlist` [24].

Finally, we mention an idea of Richard Bornat's [1], that instead of denoting a sequence of data items, a list (or other structure) should denote the sequence (or other collection) of addresses at which the data items are stored. In the case of simply linked lists, we write $\text{listN } \sigma(i, j)$ when there is

a list segment from i to j containing the sequence σ of addresses:



This view leads to the definition

$$\text{listN } \epsilon(i, j) \stackrel{\text{def}}{=} \text{emp} \wedge i = j$$

$$\text{listN } l \cdot \sigma(i, k) \stackrel{\text{def}}{=} l = i \wedge \exists j. i + 1 \mapsto j * \text{listN } \sigma(j, k).$$

Notice that listN is extremely local: It holds for a heap containing only the second cell of each record in the list structure.

The reader may verify that the body of the list-reversing program preserves the invariant

$$\exists \sigma, \delta. (\text{listN } \sigma(i, \text{nil}) * \text{listN } \delta(j, \text{nil})) \wedge \sigma_0^\dagger = \sigma^\dagger \cdot \delta,$$

where σ_0 is the original sequence of addresses of the data fields of the list at i . In fact, since it shows that these addresses do not change, this invariant embodies a stronger notion of “in-place algorithm” than that given before.

6. Trees and Dags

When we move from list to tree structures, the possible patterns of sharing within the structures become richer.

In this section, we will focus on a particular kind of abstract value called an “S-expression” in the LISP community. The set S-exp of these values is the least set such that

$$\tau \in \text{S-exp} \text{ iff }$$

$$\tau \in \text{Atoms}$$

$$\text{or } \tau = (\tau_1 \cdot \tau_2) \text{ where } \tau_1, \tau_2 \in \text{S-exp}.$$

(Of course, this is just a particular, and very simple, initial algebra. We could take carriers of any anarchic many-sorted initial algebra to be our abstract data, but this would complicate our exposition while adding little of interest.)

For clarity, it is vital to maintain the distinction between abstract values and their representations. Thus, we will call abstract values “S-expressions”, while calling representations without sharing “trees”, and representations with sharing but no loops “dags” (for directed acyclic graphs).

We write $\text{tree } \tau(i)$ (or $\text{dag } \tau(i)$) to indicate that i is the root of a tree (or dag) representing the S-expression τ . Both predicates are defined by induction on the structure of τ :

$$\text{tree } a(i) \text{ iff } \text{emp} \wedge i = a$$

$$\text{tree } (\tau_1 \cdot \tau_2)(i) \text{ iff }$$

$$\exists i_1, i_2. i \mapsto i_1, i_2 * \text{tree } \tau_1(i_1) * \text{tree } \tau_2(i_2)$$

$$\text{dag } a(i) \text{ iff } i = a$$

$$\text{dag } (\tau_1 \cdot \tau_2)(i) \text{ iff }$$

$$\exists i_1, i_2. i \mapsto i_1, i_2 * (\text{dag } \tau_1(i_1) \wedge \text{dag } \tau_2(i_2)).$$

Here, since **emp** is omitted from its definition, $\text{dag } a(i)$ is pure, and therefore intuitionistic. By induction, it is easily seen that $\text{dag } \tau i$ is intuitionistic for all τ . In fact, this is vital, since we want $\text{dag } \tau_1(i_1) \wedge \text{dag } \tau_2(i_2)$ to hold for a heap that contains the (possibly overlapping) sub-dags, but not to assert that the sub-dags are identical.

To express simple algorithms for manipulating trees, we must introduce recursive procedures. However, to avoid problems of aliased variables and interfering procedures, we will limit ourselves to first-order procedures without global variables (except, in the case of recursion, the name of procedure being defined), and we will explicitly indicate which formal parameters are modified by the procedure body. Thus a procedure definition will have the form

$$h(x_1, \dots, x_m; y_1, \dots, y_n) = c,$$

where y_1, \dots, y_n are the free variables modified by c , and x_1, \dots, x_m are the other free variables of c , except h .

We will not declare procedures at the beginning of blocks, but will simply assume that a program is reasoned about in the presence of procedure definitions. In this setting, an appropriate inference rule for partial correctness is

- Procedures

$$\text{When } h(x_1, \dots, x_m; y_1, \dots, y_n) = c,$$

$$\{p\} h(x_1, \dots, x_m; y_1, \dots, y_n) \{q\}$$

$$\vdots$$

$$\{p\} c \{q\}$$

$$\hline \{p\} h(x_1, \dots, x_m; y_1, \dots, y_n) \{q\}.$$

In essence, to prove some specification of a call of a procedure in which the actual parameters are the same as the formal parameters in the procedure definition, one proves a similar specification of the body of the definition under the *recursion hypothesis* that recursive calls satisfy the specification being proved.

Of course, one must be able to deduce specifications about calls in which the actual parameters differ from the formals. For this purpose, however, the structural inference rule for substitution suffices, if one takes the variables modified by $h(x_1, \dots, x_m; y_1, \dots, y_n)$ to be y_1, \dots, y_n . Note

that the restrictions on this rule prevent the creation of aliased variables or expressions.

For example, we would expect a tree-copying procedure

```
copytree(i; j) =
  if isatom(i) then j := i else
    newvar i1, i2, j1, j2 in
      (i1 := [i] ; i2 := [i + 1] ;
       copytree(i1; j1) ; copytree(i2; j2) ;
       j := cons(j1, j2))
```

to satisfy

$$\{ \text{tree } \tau(i) \} \text{ copytree}(i; j) \{ \text{tree } \tau(i) * \text{tree } \tau(j) \}. \quad (5)$$

The following is a proof outline of a similar specification of the procedure body:

```
{tree τ(i)}
if isatom(i) then
  {isatom(τ) ∧ emp ∧ i = τ}
  {isatom(τ) ∧ ((emp ∧ i = τ) * (emp ∧ i = τ))}
  j := i
  {isatom(τ) ∧ ((emp ∧ i = τ) * (emp ∧ j = τ))}
else
  {∃τ1, τ2. τ = (τ1 · τ2) ∧ tree (τ1 · τ2)(i)}
  newvar i1, i2, j1, j2 in
    (i1 := [i] ; i2 := [i + 1] ;
     {∃τ1, τ2. τ = (τ1 · τ2) ∧ (i ↦ i1, i2 *
       tree τ1(i1) * tree τ2(i2))})
    copytree(i1; j1);
    {∃τ1, τ2. τ = (τ1 · τ2) ∧ (i ↦ i1, i2 *
      tree τ1(i1) * tree τ2(i2) * tree τ1(j1))}
    copytree(i2; j2);
    {∃τ1, τ2. τ = (τ1 · τ2) ∧
      (i ↦ i1, i2 * tree τ1(i1) * tree τ2(i2) *
      tree τ1(j1) * tree τ2(j2))}
    j := cons(j1, j2)
    {∃τ1, τ2. τ = (τ1 · τ2) ∧
      (i ↦ i1, i2 * tree τ1(i1) * tree τ2(i2) *
      j ↦ j1, j2 * tree τ1(j1) * tree τ2(j2))}
    {∃τ1, τ2. τ = (τ1 · τ2) ∧
      (tree (τ1 · τ2)(i) * tree (τ1 · τ2)(j)))}
{tree τ(i) * tree τ(j)}.
```

To obtain the specifications of the recursive calls in this proof outline from the recursion hypothesis (5), one must use the frame rule to move from the footprint of the recursive call to the larger footprint of the procedure body. In more detail, the specification of the first recursive call in the outline can be obtained by the inferences

$$\frac{\begin{array}{c} \{ \text{tree } \tau(i) \} \text{ copytree}(i; j) \{ \text{tree } \tau(i) * \text{tree } \tau(j) \} \\ \hline \{ \text{tree } \tau_1(i_1) \} \text{ copytree}(i_1; j_1) \{ \text{tree } \tau_1(i_1) * \text{tree } \tau_1(j_1) \} \end{array}}{\frac{\begin{array}{c} \{ (\tau = (\tau_1 \cdot \tau_2) \wedge i \mapsto i_1, i_2) * \\ \text{tree } \tau_1(i_1) * \text{tree } \tau_2(i_2) \} \end{array}}{\frac{\begin{array}{c} \{ (\tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \\ \text{tree } \tau_1(i_1) * \text{tree } \tau_2(i_2))) \} \\ \hline \{ \text{copytree}(i_1; j_1); \end{array}}{\frac{\begin{array}{c} \{ (\tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \\ \text{tree } \tau_1(i_1) * \text{tree } \tau_2(i_2) * \text{tree } \tau_1(j_1))) \} \\ \hline \{ \exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \\ \text{tree } \tau_1(i_1) * \text{tree } \tau_2(i_2)) \} \end{array}}{\frac{\begin{array}{c} \{ \text{copytree}(i_1; j_1); \end{array}}{\frac{\begin{array}{c} \{ \exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \wedge (i \mapsto i_1, i_2 * \\ \text{tree } \tau_1(i_1) * \text{tree } \tau_2(i_2) * \text{tree } \tau_1(j_1)) \}, \end{array}}{}}}}}}$$

using the substitution rule, the frame rule, the rule of consequence (with the axiom that $(p \wedge q) * r \Leftrightarrow p \wedge (q * r)$ when p is pure), and auxiliary variable elimination.

What we have proved about `copytree`, however, is not as general as possible. In fact, this procedure is insensitive to sharing in its input, so that it also satisfies

$$\{ \text{dag } \tau(i) \} \text{ copytree}(i; j) \{ \text{dag } \tau(i) * \text{tree } \tau(j) \}. \quad (6)$$

But if we try to prove this specification by mimicking the previous proof, we encounter a problem: For (say) the first recursive call, at the point where we used the frame rule, we must infer

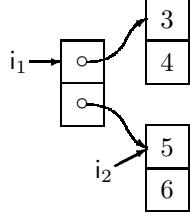
$$\begin{aligned} & \{(\dots) * (\text{dag } \tau_1(i_1) \wedge \text{dag } \tau_2(i_2))\} \\ & \text{copytree}(i_1; j_1) \\ & \{(\dots) * (\text{dag } \tau_1(i_1) \wedge \text{dag } \tau_2(i_2)) * \text{tree } \tau_1(j_1)\}. \end{aligned}$$

Now, however, the presence of \wedge instead of $*$ prevents us from using the frame rule — and for good reason: The recursion hypothesis (6) is not strong enough to imply this specification of the recursive call, since it does not imply that `copytree(i1; j1)` leaves unchanged the portion of the

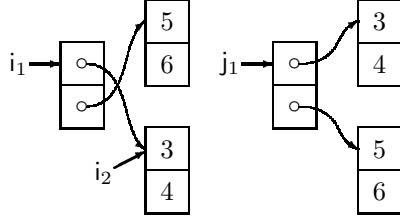
heap shared by the dags representing τ_1 and τ_2 . For example, suppose

$$\tau_1 = ((3 \cdot 4) \cdot (5 \cdot 6)) \quad \tau_2 = (5 \cdot 6).$$

Then (6) would permit $\text{copytree}(i_1; j_1)$ to change the state from



into



where dag τ_2 (i_2) is false.

One way of surmounting this problem is to extend assertions to contain *assertion variables*, and to extend the substitution rule so that the v_i 's include assertion variables as well as ordinary variables, with assertions being substituted for these assertion variables.

Then we can use an assertion variable p to specify that every property of the heap that is active before executing $\text{copytree}(i; j)$ remains true after execution:

$$\{p \wedge \text{dag } \tau(i)\} \text{ copytree}(i; j) \{p * \text{tree } \tau(j)\}. \quad (7)$$

We leave the proof outline to the reader, but show the inference of the specification of the first recursive call from the recursion hypothesis, using the substitution rule and auxiliary variable elimination:

$$\begin{array}{c} \{p \wedge \text{dag } \tau(i)\} \text{ copytree}(i; j) \{p * \text{tree } \tau(j)\} \\ \hline \{(\tau = (\tau_1 \cdot \tau_2) \wedge p \wedge \text{dag } \tau_2(i_2)) \wedge \text{dag } \tau_1(i_1)\} \\ \text{copytree}(i_1; j_1); \\ \{(\tau = (\tau_1 \cdot \tau_2) \wedge p \wedge \text{dag } \tau_2(i_2)) * \text{tree } \tau_1(j_1)\} \\ \hline \{\exists \tau_1, \tau_2. (\tau = (\tau_1 \cdot \tau_2) \wedge p \wedge \text{dag } \tau_2(i_2)) \wedge \text{dag } \tau_1(i_1)\} \\ \text{copytree}(i_1; j_1); \\ \{\exists \tau_1, \tau_2. (\tau = (\tau_1 \cdot \tau_2) \wedge p \wedge \text{dag } \tau_2(i_2)) * \text{tree } \tau_1(j_1)\}. \end{array}$$

7. Arrays and Iterated Separating Conjunction

It is straightforward to extend our programming language to include heap-allocated one-dimensional arrays, by

introducing an allocation command where the number of consecutive heap cells to be allocated is specified by an operand. It is simplest to leave the initial values of these cells indeterminate. We will use the syntax

$$\langle \text{comm} \rangle ::= \dots | \langle \text{var} \rangle := \text{allocate } \langle \text{exp} \rangle$$

with the operational semantics

$$\langle v := \text{allocate } e, (s, h) \rangle \rightsquigarrow ([s \mid v: \ell], h \cdot h')$$

where $h \perp h'$ and $\text{dom } h' = \{i \mid \ell \leq i < \ell + \llbracket e \rrbracket_{\text{exp}} s\}$.

To describe such arrays, it is helpful to extend the concept of separating conjunction to a construct that iterates over a finite consecutive set of integers. We use the syntax

$$\langle \text{assert} \rangle ::= \dots | \odot_{\langle \text{var} \rangle = \langle \text{exp} \rangle}^{\langle \text{exp} \rangle} \langle \text{assert} \rangle$$

with the meaning

$$\llbracket \odot_{v=e}^{e'} p \rrbracket_{\text{asrt}}(s, h) =$$

$$\text{let } m = \llbracket e \rrbracket_{\text{exp}} s, n = \llbracket e' \rrbracket_{\text{exp}} s,$$

$$I = \{i \mid m \leq i \leq n\} \text{ in}$$

$$\exists H \in I \rightarrow \text{Heaps.}$$

$$\forall i, j \in I. i \neq j \text{ implies } Hi \perp Hj$$

$$\text{and } h = \bigcup \{Hi \mid i \in I\}$$

$$\text{and } \forall i \in I. \llbracket p \rrbracket_{\text{asrt}}([s \mid v: i], Hi).$$

The following axiom schemata are useful:

$$m > n \Rightarrow (\odot_{i=m}^n p(i) \Leftrightarrow \text{emp})$$

$$m = n \Rightarrow (\odot_{i=m}^n p(i) \Leftrightarrow p(m))$$

$$k \leq m \leq n + 1 \Rightarrow$$

$$(\odot_{i=k}^n p(i) \Leftrightarrow (\odot_{i=k}^{m-1} p(i) * \odot_{i=m}^n p(i)))$$

$$\odot_{i=m}^n p(i) \Leftrightarrow \odot_{i=m-k}^{n-k} p(i+k)$$

$$m \leq n \Rightarrow$$

$$((\odot_{i=m}^n p(i)) * q \Leftrightarrow \odot_{i=m}^n (p(i) * q)) \text{ when } q \text{ is pure}$$

$$m \leq j \leq n \Rightarrow ((\odot_{i=m}^n p(i)) \Rightarrow (p(j) * \text{true})).$$

A simple example is the use of an array as a cyclic buffer. We assume that an n -element array has been allocated at address l , e.g., by $l := \text{allocate } n$, and we use the variables

m number of active elements

i address of first active element

j address of first inactive element.

Then when the buffer contains a sequence α , it should satisfy the assertion

$$0 \leq m \leq n \wedge l \leq i < l + n \wedge l \leq j < l + n \wedge \\ j = i \oplus m \wedge m = \#\alpha \wedge \\ ((\bigodot_{k=0}^{m-1} i \oplus k \mapsto \alpha_{k+1}) * (\bigodot_{k=0}^{n-m-1} j \oplus k \mapsto -)),$$

where $x \oplus y = x + y$ modulo n , and $l \leq x \oplus y < l + n$.

Somewhat surprisingly, iterated separating conjunction is useful for making assertions about structures that do not involve arrays. For example, the connection between our list and Bornat's listN is given by

$$\text{list } \alpha(i, j) \Leftrightarrow$$

$$\exists \sigma. \# \sigma = \#\alpha \wedge (\text{listN } \sigma(i, j) * \bigodot_{k=1}^{\#\alpha} \sigma_k \mapsto \alpha_k).$$

A more elaborate example is provided by a program that accepts a list i representing a sequence α of integers, and produces a list j of lists representing all (not necessarily contiguous) subsequences of α . This program is a variant of a venerable LISP example that has often been used to illustrate the storage economy that can be obtained in LISP by a straightforward use of sharing. Our present interest is in using iterated separating conjunction to specify the sharing pattern in sufficient detail to show the amount of storage that is used.

First, given a sequence σ of sequences, we define $\text{ext}_a \sigma$ to be the sequence of sequences obtained by prefixing the integer a to each sequence in σ :

$$\begin{aligned} \#\text{ext}_a \sigma &\stackrel{\text{def}}{=} \#\sigma \\ (\text{ext}_a \sigma)_i &\stackrel{\text{def}}{=} a \cdot \sigma_i. \end{aligned}$$

Then we define $\text{ss}(\alpha, \sigma)$ to assert that σ is a sequence of the subsequences of α (in the particular order that is produced by an iterative program):

$$\begin{aligned} \text{ss}(\epsilon, \sigma) &\stackrel{\text{def}}{=} \sigma = [\epsilon] \\ \text{ss}(a \cdot \alpha, \sigma) &\stackrel{\text{def}}{=} \exists \sigma'. \text{ss}(\alpha, \sigma') \wedge \sigma = (\text{ext}_a \sigma')^\dagger \cdot \sigma'. \end{aligned}$$

(To obtain the different order produced by a simple recursive program, we would remove the reflection (\dagger) operator here and later.) Next, we define $Q(\sigma, \beta)$ to assert that β is a sequence of lists whose components represent the components of σ :

$$Q(\sigma, \beta) \stackrel{\text{def}}{=} \#\beta = \#\sigma \wedge \forall_{i=1}^{\#\beta} (\text{list } \sigma_i(\beta_i, \text{nil}) * \text{true}).$$

At this stage, we can specify the abstract behavior of the subsequence program subseq by

$$\begin{aligned} &\{\text{list } \alpha(i, \text{nil})\} \\ &\text{subseq} \\ &\{\exists \sigma, \beta. \text{ss}(\alpha^\dagger, \sigma) \wedge (\text{list } \beta(j, \text{nil}) * (Q(\sigma, \beta)))\}. \end{aligned}$$

To capture the sharing structure, however, we use iterated separating conjunction to define $R(\beta)$ to assert that the last element of β is the empty list, while every previous element consists of a single integer cons'ed onto some later element of β :

$$\begin{aligned} R(\beta) &\stackrel{\text{def}}{=} (\beta_{\#\beta} = \text{nil} \wedge \text{emp}) * \\ &\quad \bigodot_{i=1}^{\#\beta-1} (\exists a, k. i < k \leq \#\beta \wedge \beta_i \mapsto a, \beta_k). \end{aligned}$$

Here the heap described by each component of the iteration contains a single cons-pair, so that the heap described by $R(\beta)$ contains $\#\beta - 1$ cons-pairs. Finally, the full specification of c is

$$\begin{aligned} &\{\text{list } \alpha(i, \text{nil})\} \\ &\text{subseq} \\ &\{\exists \sigma, \beta. \text{ss}(\alpha^\dagger, \sigma) \wedge (\text{list } \beta(j, \text{nil}) * (Q(\sigma, \beta) \wedge R(\beta)))\}. \end{aligned}$$

Here the heap described by $\text{list } \beta(j, \text{nil})$ contains $\#\beta$ cons-pairs, so that the entire heap described by the postcondition contains $(2 \times \#\beta) - 1 = (2 \times 2^{\#\alpha}) - 1$ cons-pairs.

8. Proving the Schorr-Waite Algorithm

The most ambitious application of separation logic has been Yang's proof of the Schorr-Waite algorithm for marking structures that contain sharing and cycles [33, 34]. This proof uses the older form of classical separation logic [19] in which address arithmetic is forbidden and the heap maps addresses into multifield records — each containing, in this case, two address fields and two boolean fields.

Several significant features of the proof are evident from its main invariant:

$$\begin{aligned} &\text{noDanglingR} \wedge \text{noDangling(t)} \wedge \text{noDangling(p)} \wedge \\ &(\text{listMarkedNodesR(stack, p)} * \\ &(\text{restoredListR(stack, t)} \rightarrow \text{spansR(STree, root)})) \wedge \\ &(\text{markedR} * (\text{unmarkedR} \wedge (\forall x. \text{allocated}(x) \Rightarrow \\ &(\text{reach}(t, x) \vee \text{reachRightChildInList}(stack, x))))). \end{aligned}$$

The heap described by this invariant is the footprint of the entire algorithm, which is exactly the structure that is reachable from the address root. Since addresses now refer to entire records, it is easy to assert that the record at x has been allocated:

$$\text{allocated}(x) \stackrel{\text{def}}{=} x \hookrightarrow -, -, -, -, -,$$

that all active records are marked:

$$\text{markedR} \stackrel{\text{def}}{=} \forall x. \text{allocated}(x) \Rightarrow x \hookrightarrow -, -, -, \text{true},$$

that x is not a dangling address:

$$\text{noDangling}(x) \stackrel{\text{def}}{=} (x = \text{nil}) \vee \text{allocated}(x),$$

or that no active record contains a dangling address:

$$\begin{aligned} \text{noDanglingR} &\stackrel{\text{def}}{=} \forall x, l, r. (x \hookleftarrow l, r, -, -) \Rightarrow \\ &\quad \text{noDangling}(l) \wedge \text{noDangling}(r). \end{aligned}$$

(Yang uses the helpful convention that the predicates with names ending in “R” are those that are not intuitionistic.)

In the second line of the invariant, the subassertion $\text{listMarkedNodesR}(\text{stack}, p)$ holds for a part of the heap called the *spine*, which is a linked list of records from root to the current address t in which the links have been reversed; the variable stack is a sequence of four-tuples determining the contents of the spine. In the next line, $\text{restoredListR}(\text{stack}, t)$ describes what the contents of the spine should be after the links have been restored, and $\text{spansR}(\text{STree}, \text{root})$ asserts that the abstract structure STree is a spanning tree of the heap.

The assertion $\text{spansR}(\text{STree}, \text{root})$ also appears in the precondition of the algorithm. Thus, the second and third lines of the invariant use separating implication elegantly to assert that, if the spine is correctly restored, then the heap will have the same spanning tree as it had initially. (In fact, the proof goes through if $\text{spansR}(\text{STree}, \text{root})$ is any predicate that is independent of the boolean fields in the records; spanning trees are used only because they are enough to determine the heap, except for the boolean fields.) To the author’s knowledge, this part of the invariant is the only conceptual use of separating implication in a real proof (as opposed to its formal use in expressing weakest preconditions).

In the rest of the invariant, the heap is partitioned into marked and unmarked records, and it is asserted that every active unmarked record can be reached from the variable t or from certain fields in the spine. However, since this assertion lies within the right operand of the separating conjunction that separates marked and unmarked notes, the paths by which the unmarked records are reached must consist of unmarked records. Anyone (such as the author [27, Section 5.1]) who has tried to verify this kind of graph traversal, even informally, will appreciate the extraordinary succinctness of the last two lines of Yang’s invariant.

9. Computability and Complexity Results

The existence of weakest preconditions for each of our new commands assures us that a central property of Hoare logic is preserved by our extension: that a program specification annotated with loop invariants and recursion hypotheses (and, for total correctness, variants) can be reduced

to a collection of assertions, called verification conditions, whose validity will insure the original specification. Thus the central question for computability and complexity is to decide the validity of assertions in separation logic.

Yang [8, 34] has examined the decidability of classical separation logic without arithmetic (where expressions are variables, values are addresses or **nil**, and the heap maps addresses into two-field records). He showed that, even when the characteristic operations of separation logic (**emp**, \hookrightarrow , $*$, and $-*$, but not \hookleftarrow) are prohibited, deciding the validity of an assertion is not recursively enumerable. (As a consequence, we cannot hope to find an axiomatic description of \hookrightarrow .) On the other hand, Yang and Calcagno showed that if the characteristic operations are permitted but quantifiers are prohibited, then the validity of assertions is algorithmically decidable.

For the latter case, Calcagno and Yang [8] have investigated complexity. Specifically, for the languages tabulated below they considered

MC The model checking problem: Does $\llbracket p \rrbracket_{\text{asrt}} sh$ hold for a specified state (s, h) ?

VAL The validity problem: Does $\llbracket p \rrbracket_{\text{asrt}} sh$ hold for all states (s, h) ?

In each case, they determined that the problem was complete for the indicated complexity class:

	Language	MC	VAL
\mathcal{L}	$P ::= E \hookrightarrow E, E \mid \neg E \hookleftarrow -, -$ $E = E \mid E \neq E \mid \text{false}$ $P \wedge P \mid P \vee P \mid \text{emp}$	P	coNP
\mathcal{L}^*	$P ::= \mathcal{L} \mid P * P$	NP	Π_2^P
$\mathcal{L}^{\neg*}$	$P ::= \mathcal{L} \mid \neg P \mid P * P$		PSPACE
\mathcal{L}^{*-}	$P ::= \mathcal{L} \mid P -* P$		PSPACE
$\mathcal{L}^{\neg*\neg*}$	$P ::= \mathcal{L} \mid \neg P \mid P * P \mid P -* P$		PSPACE

10. Garbage Collection

Since our logic permits programs to use unrestricted address arithmetic, there is little hope of constructing any general-purpose garbage collector. On the other hand, the situation for the older logic, in which addresses are disjoint from integers, is more hopeful. However, it is clear that this logic permits one to make assertions, such as “The heap contains two elements” that might be falsified by the execution of a garbage collector, even though, in any realistic sense, such an execution is unobservable.

The present author [28] has given the following example

(shown here as a proof outline):

```

{true}
x := cons(3, 4);
{x ↪ 3, 4}
{∃x. x ↪ 3, 4}
x := nil
{∃x. x ↪ 3, 4},

```

where the final assertion describes a disconnected piece of structure, and would be falsified if the disconnected piece were reclaimed by a garbage collector. In this case, the assertions are all intuitionistic, and indeed it is hard to concoct a reasonable program logic that would prohibit such a derivation.

Calcagno, O’Hearn, and Bornat [7, 6, 5, 4] have explored ways of avoiding this problem by defining the existential quantifier in a nonstandard way, and have defined a rich variety of logics that are insensitive to garbage collection. Unfortunately, there is no brief way of relating these logics to those discussed in this paper.

11. Future Directions

11.1. New Forms of Inference

In Yang’s proof of the Schorr-Waite algorithm, there are thirteen assertions that have been semantically validated but do not seem to be consequences of known inference rules, and are far too specific to be considered axioms. This surprising state of affairs is likely a consequence of the novel character of the proof itself — especially the quantification over all allocated addresses, and the crucial use of separating implication — as well as the fact that the algorithm deals with sharing in a more fundamental way than others that have been studied with separation logic.

The generalization of such assertions may be a fertile source of new inference rules. For example, suppose \hat{p} is intuitionistic, and let p be $\forall x. \text{allocated}(x) \Rightarrow \hat{p}$. Then

$$\begin{aligned}
& \text{emp} \Rightarrow p \\
& (\exists x. (x \mapsto -, -, -, -) \wedge \hat{p}) \Rightarrow p \\
& (p * p) \Rightarrow p.
\end{aligned}$$

For a more elaborate example, suppose we say that two assertions are *immiscible* if they cannot both hold for overlapping heaps. More precisely, p and q are immiscible iff, for all stores s and heaps h and h' such that $h \cup h'$ is a function, if $\llbracket p \rrbracket_{\text{asrt}} sh$ and $\llbracket q \rrbracket_{\text{asrt}} sh'$ then $h \perp h'$.

On the one hand, it is easy to find immiscible assertions: If \hat{p} and \hat{q} are intuitionistic and $\hat{p} \wedge \hat{q} \Rightarrow \text{false}$ is valid, then

$$\forall x. \text{allocated}(x) \Rightarrow \hat{p} \quad \text{and} \quad \forall x. \text{allocated}(x) \Rightarrow \hat{q}$$

are immiscible. Moreover, if p' and q' are immiscible and $p \Rightarrow p'$ and $q \Rightarrow q'$ are valid, then p and q are immiscible.

On the other hand, if p and q are immiscible, then

$$(p * \text{true}) \wedge (q * r) \Rightarrow q * ((p * \text{true}) \wedge r)$$

is valid.

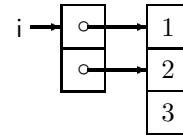
Both of these examples are sound, and useful for at least one proof of a specification of a real program. But they may well be special cases of more generally useful rules or other inference mechanisms. O’Hearn suspects that there are good prospects to find a useful proof theory for separation logic using “labeled deduction” [13, 14].

It should also be noted that Yang’s proof depends critically on the fact that the Schorr-Waite algorithm performs an in-place computation. New problems may arise in trying to prove, say, a program that copies a reachable structure while preserving its sharing patterns — somehow, one must express the isomorphism between the structure and its copy.

Beyond these particulars, in its present state separation logic is not only theoretically incomplete, but *practically* incomplete: As it is applied in new ways, there will be a need for new kinds of inference.

11.2. Taming Address Arithmetic

When we first realized that separation logic could be generalized and simplified by permitting address arithmetic, it seemed an unalloyed benefit. But there are problems. For example, consider the definition of dag given in Section 6: The assertion $\text{dag}((1 \cdot 2) \cdot (2 \cdot 3)) \text{ (i)}$ holds in states where two distinct records overlap, e.g.



Similarly, iff one were to try to recast the Schorr-Waite proof in the logic with address arithmetic, it would be difficult (but necessary) to assert that distinct records do not overlap, and that all address values in the program denote the first fields of records.

These problems, of what might be called *skewed sharing*, become even more difficult when there are several types of records, with differing lengths and field types.

A possible solution may be to augment states with a mapping from the domain of the heap into “attributes” that could be set by the program, described by assertions, and perhaps tested to determine whether to abort. These attributes would be similar to auxiliary variables (in the sense of Owicky and Gries [25]), in that their values could not influence the values of nonauxiliary variables or heap cells, nor the flow of control.

To redefine **dag** to avoid skewed sharing, one could, at each allocation $x := \mathbf{cons}(e_1, e_2)$ that creates a record in a dag, give x (but not $x + 1$) the attribute **dag-record**. Then the definition of a non-atomic dag would be changed to

$$\begin{aligned} \mathbf{dag}(\tau_1 \cdot \tau_2)(i) \text{ iff } & \mathbf{is-dag-record}(i) \wedge \\ & \exists i_1, i_2. \ i \mapsto i_1, i_2 * (\mathbf{dag} \tau_1(i_1) \wedge \mathbf{dag} \tau_2(i_2)). \end{aligned}$$

In a version of the Schorr-Waite proof using address arithmetic, one might give the attribute **record** to the address of the beginning of each record. Then one would add

$$\begin{aligned} \forall x. \ \mathbf{is-record}(x) \Rightarrow & \exists l, r, c, m. \ x \hookrightarrow l, r, c, m \\ & \wedge (\mathbf{is-record}(l) \vee l = \mathbf{nil}) \wedge (\mathbf{is-record}(r) \vee r = \mathbf{nil}) \\ & \wedge \mathbf{is-boolean}(c) \wedge \mathbf{is-boolean}(m) \end{aligned}$$

to the invariant, and use the assertion $\mathbf{is-record}(x)$ in place of $\mathbf{allocated}(x)$.

There is a distinct flavor of types in this use of attributes: The attributes record information, for purposes of proof, that in a typed programming language would be checked by the compiler and discarded before runtime. An alternative, of course, would be to move to a typed programming language, but our goal is to keep the programming language low-level and, for simplicity and flexibility, to use the proof system to replace types rather than supplement them.

11.3. Concurrency

In the 1970's, Hoare and Brinch Hansen argued persuasively that, to prevent data races where two processes attempt to access the same storage without synchronization, concurrent programs should be syntactically restricted to limit process interference to explicitly indicated *critical regions*. In the presence of shared mutable structures, however, processes can interfere in ways too subtle to be detected syntactically.

On the other hand, when one turns to program verification, it is clear that separation logic can specify the absence of interference. In the simplest situation, the concurrent execution $c_1 \parallel c_2$ of two processes that do not interfere with one another is described by the inference rule

$$\frac{\{p_1\} \ c_1 \ \{q_1\} \quad \{p_2\} \ c_2 \ \{q_2\}}{\{p_1 * p_2\} \ c_1 \parallel c_2 \ \{q_1 * q_2\}},$$

where no variable free in p_1 or q_1 is modified by c_2 , or vice-versa.

Going beyond this trivial case, O'Hearn [22, 21] has extended the Hoare-like logic for concurrency devised by Owicki and Gries [25] (which is in turn an extension of work by Hoare [18]), to treat critical regions in the framework of separation logic.

The basic idea is that, just as program variables are syntactically partitioned into groups owned by different processes and resources, so the heap should be similarly partitioned by separating conjunctions in the proof of the program. The most interesting aspect is that the partition of the heap can be changed by executing critical regions associated with resources, so that ownership of a particular address can move from a process to a resource and from there to another process.

Thus, for example, one process may allocate addresses and place them in a buffer, while another process removes the addresses from the buffer and deallocates them. Similarly, in a concurrent version of quicksort, an array segment might be divided between two concurrent recursive calls, and reunited afterwards.

Unfortunately, at this writing there is no proof that O'Hearn's inference rules are sound. The difficulty is that, in O'Hearn's words, "Ownership is in the eye of the asserter", i.e., the changing partitions of the heap are not determined by the program itself, but only by the assertions in its proof.

In concurrent programming, it is common to permit several processes to read the same variable, as long as no process can modify its value simultaneously. It would be natural and useful to extend this notion of *passivity* to heap cells, so that the specification of a process might indicate that a portion of the heap is evaluated but never mutated, allocated or deallocated by the process. (This capability would also provide an alternative way to specify the action of *copytree* on dags discussed in Section 6.) Semantically, this would likely require activity bits to take on a third, intermediate value of "read-only".

Concurrency often changes the focus from terminating programs to programs that usefully run on forever — for which Hoare logic is of limited usefulness. For such programs, it might be helpful to extend temporal logic with separating operators.

11.4. Storage Allocation

Since separation logic describes a programming language with explicit allocation and deallocation of storage, without any behind-the-scenes garbage collector, it should be suitable for reasoning about allocation methods themselves.

A challenging example is the use of *regions*, in the sense of Tofte et al [31]. To provide an explicitly programmable region facility, one must program an allocator and deallocator for regions of storage, each of which is equipped with its own allocator and deallocator. Then one must prove that the program using these routines never deallocates a region unless it is safe to deallocate all storage that has been allocated from that region.

Although separation logic is incompatible with general-purpose garbage collection (at least when unrestricted address arithmetic is permitted), it should be possible to construct and verify a garbage collector for a specific program. In this situation, one should be able to be far more aggressive than would be possible for a general-purpose garbage collector, both in recovering storage, and in minimizing the extra data needed to guide the traversal of active structure. For example, for the representation described by dag in Section 6, it would be permissible for the collector to increase the amount of sharing.

The key here is that a correct garbage collector need only maintain the assertion that must hold at the point where the collector is called, while preserving the value of certain input variables that determine the abstract values being computed. (In addition, for total correctness, the collector must not increase the value of the loop variants that insure termination.) For instance, in the list-reversal example in Section 5, a collector called at the end of the **while** body would be required to maintain the invariant

$$\exists \alpha, \beta. (\text{list } \alpha(i, \text{nil}) * \text{list } \beta(j, \text{nil})) \wedge \alpha_0^\dagger = \alpha^\dagger \cdot \beta,$$

while preserving the abstract input α_0 , and not increasing the variant, which is the length of α .

It is interesting to note that the garbage collector is required to respect the partial equivalence relation determined by the invariant, in which two states are equivalent when they both satisfy the invariant and specify the same values of the input variables. Considering the use of partial equivalence relations to give semantics to types, this reinforces the view that types and assertions are semantically similar.

11.5. The Relationship to Type Systems

Although types and assertions may be semantically similar, the actual development of type systems for programming languages has been quite separate from the development of approaches to specification such as Hoare logic, refinement calculi, or model checking. In particular, the idea that states have types, and that executing a command may change the type of a state, has only taken hold recently, in the study of type systems for low-level languages, such as the *alias types* devised by Walker and Morrisett [32].

Separation logic is closely related to such type systems. Indeed, their commonality can be captured roughly by a simple type system:

```
 $\langle \text{value type} \rangle ::= \text{integer} \mid \langle \text{address variable} \rangle$ 
 $\langle \text{store type} \rangle ::=$ 
 $\langle \text{variable} \rangle : \langle \text{value type} \rangle, \dots, \langle \text{variable} \rangle : \langle \text{value type} \rangle$ 
```

```
 $\langle \text{heap type} \rangle ::= \text{emp}$ 
 $\mid \langle \text{address variable} \rangle \mapsto \langle \text{value type} \rangle, \dots, \langle \text{value type} \rangle$ 
 $\mid \langle \text{heap type} \rangle * \langle \text{heap type} \rangle$ 
```

 $\langle \text{state type} \rangle ::= \langle \text{store type} \rangle ; \langle \text{heap type} \rangle$

It is straightforward to translate state types in this system into assertions of classical separation logic (in the formulation without address arithmetic) or into alias types.

It may well be possible to devise a richer type system that accommodates a limited degree of address arithmetic, permitting, for example, addresses that point into the interior of records, or relative addresses.

Basically, however, the real question is whether the dividing line between types and assertions can be erased.

11.6. Embedded Code Pointers

Even as a low-level language, the simple imperative language axiomatized by Hoare is deficient in making no provision for the occurrence in data structures of addresses that refer to machine code. Such code pointers appear in the compiled translation of programs in higher-order languages such as Scheme or SML, or object-oriented languages such as Java or C#. Moreover, they also appear in low-level programs that use the techniques of higher-order or object-oriented programming.

Yet they are difficult to describe in the first-order world of Hoare logic; to deal with embedded code pointers, we must free separation logic from both Hoare logic and the simple imperative language. Evidence of where to go comes from the recent success of type theorists in extending types to machine language (in particular to the output of compilers) [20, 32]. They have found that a higher-order functional language (with side-effects) comes to resemble a low-order machine language when two restrictions are imposed:

- Continuation-passing style (CPS) is used, so that functions receive their return addresses in the same way as they receive other parameters.
- Free variables are prohibited in expressions that denote functions, so that functions can be represented by code pointers, rather than by closures that pair code pointers with data.

In fact, this restricted language is formally similar to an imperative language in which programs are “flat”, i.e., control paths never come together except by jumping to a common label.

This raises the question of how to marry separation logic with CPS (with or without the prohibition of free variables in function expressions, which appears to be irrelevant from a logical point of view).

A simple example of CPS is provided by the function `append(x, y, r)`, which appends the list `x` to the list `y` (without mutation, so that the input lists are unchanged), and passes the resulting list on to the continuation `r`:

```
letrec append(x, y, r) = if x = nil then r(y) else
  let a = [x], b = [x + 1],
  k(z) = let w = cons(a, z) in r(w)
  in append(b, y, k).
```

We believe that the key to specifying such a CPS program is to introduce a *reflection* operator `#` that allows CPS terms to occur within assertions (in a manner reminiscent of dynamic logic [15]). Specifically, if t is a CPS term then $\# t$ is an assertion that holds for a state if t never aborts when executed in that state — in this situation we say that it is *safe* to execute t in that state.

Suppose c is a command satisfying the Hoare specification $\{p\} c \{q\}$, and t is a CPS term such that executing t has the same effect as executing c and then calling the continuation $r(x_1, \dots, x_n)$. Then we can specify t by the assertion

$$(p * (\forall x_1, \dots, x_m. q \rightarrow \# r(x_1, \dots, x_n))) \Rightarrow \# t$$

(where x_1, \dots, x_m is a subset of the variables x_1, \dots, x_n). If we ignore the difference between the separating operators and ordinary conjunction and implication, this asserts that it is safe to execute t in any state satisfying p and mapping r into a procedure such that it is safe to execute $r(x_1, \dots, x_n)$ in a state satisfying q . Taking into account the separative nature of $*$ and \rightarrow , it asserts that it is safe to execute t in any state where part of the heap satisfies p , and r is mapped into a procedure such that it is safe to execute $r(x_1, \dots, x_n)$ when the part of the heap that satisfied p is replaced by a part that satisfies q . Finally, the universal quantifier indicates the variables whose value may be changed by t before it executes $r(x_1, \dots, x_n)$.

For example, the CPS form of `append` can be specified by

$$\begin{aligned} & ((\text{list } \alpha(x, \text{nil}) * \text{list } \beta(y, \text{nil})) \\ & * (\forall z. (\text{list } \alpha(x, \text{nil}) * \text{list } \alpha(z, y) * \text{list } \beta(y, \text{nil})) \\ & \quad \rightarrow \# r(z))) \\ & \Rightarrow \# \text{append}(x, y, r). \end{aligned}$$

It can be discouraging to go back and read the “future directions” sections of old papers, and to realize how few of the future directions have been successfully pursued. It will be fortunate if half of the ideas and suggestions in this section bear fruit. In the meantime, however, the field is young, the game’s afoot, and the possibilities are tantalizing.

Acknowledgements

For encouragement and numerous suggestions, I am indebted to many researchers in separation logic, as well as the students in courses I have taught on the subject. I’m particularly grateful to Peter O’Hearn for many helpful comments after reading a preliminary draft of this paper.

However, most of the opinions herein, and all of the errors, are my own.

References

- [1] R. Bornat. Explicit description in BI pointer logic. Unpublished, June 2001.
- [2] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.
- [3] L. Caires and L. Monteiro. Verifiable and executable specifications of concurrent objects in \mathcal{L}_π . In C. Hankin, editor, *Programming Languages and Systems — ESOP ’98*, volume 1381 of *Lecture Notes in Computer Science*, pages 42–56, Berlin, 1998. Springer-Verlag.
- [4] C. Calcagno. Program logics in the presence of garbage collection (abstract). In F. Henglein, J. Hughes, H. Makhholm, and H. Niss, editors, *SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, page 11. IT University of Copenhagen, 2001.
- [5] C. Calcagno. *Semantic and Logical Properties of Stateful Programming*. Ph. D. dissertation, Università di Genova, Genova, Italy, 2002.
- [6] C. Calcagno and P. W. O’Hearn. On garbage and program logic. In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 137–151, Berlin, 2001. Springer-Verlag.
- [7] C. Calcagno, P. W. O’Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. Submitted July 2001, 2001.
- [8] C. Calcagno, H. Yang, and P. W. O’Hearn. Computability and complexity results for a spatial assertion language for data structures. In R. Hariharan, M. Mukund, and V. Vinay, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119, Berlin, 2001. Springer-Verlag.
- [9] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In M. Hennessy and P. Widmayer, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, Berlin, 2002. Springer-Verlag.
- [10] L. Cardelli and G. Ghelli. A query language based on the ambient logic. In D. Sands, editor, *Programming Languages and Systems — ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 1–22, Berlin, 2001. Springer-Verlag.
- [11] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Conference Record of POPL*

- '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 365–377, New York, 2000. ACM.
- [12] G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. The query language TQL. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*, Madison, Wisconsin, 2002.
 - [13] D. Galmiche and D. Méry. Proof-search and countermodel generation in propositional BI logic. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 263–282, Berlin, 2001. Springer-Verlag.
 - [14] D. Galmiche, D. Méry, and D. J. Pym. Resource tableaux. Submitted, 2002.
 - [15] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, Massachusetts, 2000.
 - [16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.
 - [17] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.
 - [18] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71, London, 1972. Academic Press.
 - [19] S. Ishtiaq and P. W. O’Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26, New York, 2001. ACM.
 - [20] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In X. Leroy and A. Ohori, editors, *Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Berlin, 1998. Springer-Verlag.
 - [21] P. W. O’Hearn. Notes on conditional critical regions in spatial pointer logic. Unpublished, August 31, 2001.
 - [22] P. W. O’Hearn. Notes on separation logic for shared-variable concurrency. Unpublished, January 3, 2002.
 - [23] P. W. O’Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.
 - [24] P. W. O’Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, 2001. Springer-Verlag.
 - [25] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.
 - [26] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Kluwer Academic Publishers, Boston, Massachusetts, 2002. (to appear).
 - [27] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981.
 - [28] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.
 - [29] J. C. Reynolds. Lectures on reasoning about shared mutable data structure. IFIP Working Group 2.3 School/Seminar on State-of-the-Art Program Design Using Logic, Tandil, Argentina, September 6–13, 2000.
 - [30] J. C. Reynolds and P. W. O’Hearn. Reasoning about shared mutable data structure (abstract of invited lecture). In F. Henglein, J. Hughes, H. Makhm, and H. Niss, editors, *SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, page 7. IT University of Copenhagen, 2001. The slides for this lecture are available at <ftp://ftp.cs.cmu.edu/user/jcr/spacetalk.ps.gz>.
 - [31] M. Toft and J.-P. Talpin. Implementation of the typed call-by-value λ -calculus using a stack of regions. In *Conference Record of POPL ’94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, New York, 1994. ACM Press.
 - [32] D. Walker and G. Morrisett. Alias types for recursive data structures. In R. W. Harper, editor, *Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206, Berlin, 2001. Springer-Verlag.
 - [33] H. Yang. An example of local reasoning in BI pointer logic: The Schorr-Waite graph marking algorithm. In F. Henglein, J. Hughes, H. Makhm, and H. Niss, editors, *SPACE 2001: Informal Proceedings of Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, pages 41–68. IT University of Copenhagen, 2001.
 - [34] H. Yang. *Local Reasoning for Stateful Programs*. Ph. D. dissertation, University of Illinois, Urbana-Champaign, Illinois, July 2001.
 - [35] H. Yang and P. W. O’Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416, Berlin, 2002. Springer-Verlag.

Modular Typestate Checking of Aliased Objects

Kevin Bierhoff Jonathan Aldrich

Institute for Software Research, School of Computer Science
Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
`{kevin.bierhoff.jonathan.aldrich} @ cs.cmu.edu`

Abstract

Objects often define usage protocols that clients must follow in order for these objects to work properly. Aliasing makes it notoriously difficult to check whether clients and implementations are compliant with such protocols. Accordingly, existing approaches either operate globally or severely restrict aliasing.

We have developed a sound modular protocol checking approach, based on typestates, that allows a great deal of flexibility in aliasing while guaranteeing the absence of protocol violations at runtime. The main technical contribution is a novel abstraction, *access permissions*, that combines typestate and object aliasing information. In our methodology, developers express their protocol design intent through annotations based on access permissions. Our checking approach then tracks permissions through method implementations. For each object reference the checker keeps track of the degree of possible aliasing and is appropriately conservative in reasoning about that reference. This helps developers account for object manipulations that may occur through aliases. The checking approach handles inheritance in a novel way, giving subclasses more flexibility in method overriding. Case studies on Java iterators and streams provide evidence that access permissions can model realistic protocols, and protocol checking based on access permissions can be used to reason precisely about the protocols that arise in practice.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification.

Keywords Typestates, aliasing, permissions, linear logic, behavioral subtyping.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-786-5/07/0010...\$5.00

1. Introduction

In object-oriented software, objects often define *usage protocols* that clients must follow in order for these objects to work properly. Protocols essentially define legal sequences of method calls. In conventional object-oriented languages, developers have three ways of finding out about protocols: reading informal documentation, receiving runtime exceptions that indicate protocol violations, or observing incorrect program behavior as a result of protocol violations that broke internal invariants.

It is the goal of this work to help developers follow protocols while they write code as well as to allow them to correctly and concisely document protocols for their code. We build on our previous work on leveraging *typestates* [34] for lightweight object protocol specification [4]. Our protocols are state machines that are reminiscent of Statecharts [20].

Aliasing, i.e. the existence of multiple references to the same object, is a significant complication in checking whether clients observe a protocol: a client does not necessarily know whether its reference to an object is the only reference that is active at a particular execution point. This also makes it difficult to check whether a class implements its specified protocol because reentrant callbacks through aliases can again lead to unexpected state changes.

Existing protocol checking approaches fall into two categories. They either operate globally, i.e. check an entire code base at once, or severely restrict aliasing. Global analyses typically account for aliasing but they are not suitable for interactive use during development. Moreover, they do not check whether a declared protocol is implemented correctly, a crucial requirement in object-oriented software where any class might have a protocol of its own.

Modular protocol checkers, like Fugue [12], the first sound modular typestate checker for an object-oriented language, better support developers while they write code: like a typechecker, they check each method separately for protocol violations while assuming the rest of the system to behave as specified. The trade-off, unfortunately, has been that modular checkers require code to follow pre-defined patterns of aliasing. Once a program leaves the realm of supported aliasing, any further state changes are forbidden. Generally speaking, state changes are only allowed where the checker is aware of *all* references to the changing object.

This approach has serious drawbacks. First, many examples of realistic code might be excluded. Moreover, from a developer’s point of view, the boundaries of what a checker supports are hard to predict and they might not fit with the best implementation strategy for a particular problem. Finally, aliasing restrictions arguably leave developers alone just when they have the most trouble in reasoning about their code, namely, in the presence of subtle aliasing.

This paper proposes a sound modular typestate checking approach for Java-like object-oriented languages that allows a great deal of flexibility in aliasing. For each reference, it tracks the degree of possible aliasing, and is appropriately conservative in reasoning about that reference. This helps developers account for object manipulations that may occur through aliases. High precision in tracking effects of possible aliases together with systematic support for *dynamic state tests*, i.e. runtime tests on the state of objects, make this approach feasible. Our approach helped expose a way of breaking an internal invariant that causes a commonly used Java standard library class, `java.io.BufferedInputStream`, to access an array outside its bounds. Contributions of this paper include the following.

- Our main technical contribution is a novel abstraction, called *access permissions*, that combines typestate with aliasing information about objects. Developers use access permissions to express the *design intent* of their protocols in annotations on methods and classes. Our modular checking approach verifies that implementations follow this design intent.

Access permissions systematically capture different patterns of aliasing (figure 1). A permission tracks (a) how a reference is allowed to read and/or modify the referenced object, (b) how the object might be accessed through other references, and (c) what is currently known about the object’s typestate.

- In particular, our full and pure permissions [3] capture the situation where one reference has exclusive write access to an object (a full permission) while other references are only allowed to read from the same object (using pure permissions). Read-only access through pure permissions is intuitively harmless but has to our knowledge not been exploited in existing modular protocol checkers.

- In order to increase precision of access permissions, we include two additional novel features, which make *weak permissions* more useful than in existing work. We call permissions “weak” if the referenced object can potentially be modified through other permissions.

- *Temporary state information* can be associated with weak permissions. Our checking approach makes sure that temporary state information is “forgotten” when it becomes outdated.

- Permissions can be confined to a particular part of the referenced object’s state. This allows separate permissions to independent parts of the same object. It

Access through other permissions	Current permission has ...	
	Read/write access	Read-only access
None	unique [6]	–
Read-only	full [3]	immutable [6]
Read/write	share [12]	pure [3]

Figure 1. Access permission taxonomy

also implies a *state guarantee* even for weak permissions, i.e. a guarantee that the referenced object will not leave a certain state.

- We handle inheritance in a novel way, giving subclasses more flexibility in method overriding. This is necessary for handling realistic examples of inheritance such as Java’s `BufferedInputStream` (details in section 3.2).
- We validated our approach with two case studies, iterators (section 2) and streams (section 3) from Sun’s Java standard library implementation. These case studies provide evidence that access permissions can model realistic protocols, and protocol checking based on access permissions can be used to reason precisely about the protocols that arise in practice.

A more complete evaluation of our approach is beyond the scope of this paper, which focuses on fully presenting our checking technique. The evaluation does establish that our—compared to full-fledged program verification systems [26, 2]—relatively simple approach can verify code idioms and find errors that no other decidable modular system can. The case studies reflect actual Java standard library protocols and, as far as we can tell, cannot be handled by any existing modular protocol verification system.

The following two sections introduce access permissions and verification approach with examples from our case studies before sections 4 and 5 give a formal account of our approach. Section 6 compares our approach to related work.

2. Read-Only Iterators

This section illustrates basic protocol specification and verification using our approach based on a previous case study on Java *iterators* [3]. Iterators follow a straightforward protocol but define complicated aliasing restrictions that are easily violated by developers. They are therefore a good vehicle to introduce our approach to handling aliasing in protocol verification. Iterators as presented here cannot be handled by existing modular typestate checkers due to their aliasing restrictions.

2.1 Specification Goals

The specification presented in this section models the `Iterator` interface defined in the Java standard library. For the sake of brevity we focus on *read-only* iterators, i.e. iterators that cannot modify the collection on which they iterate. We will refer to read-only iterators simply as “iterators” and qualify full Java iterators as “modifying iterators”. In earlier work we showed how to capture full Java iterators [3]. Goals of the presented specification include the following.

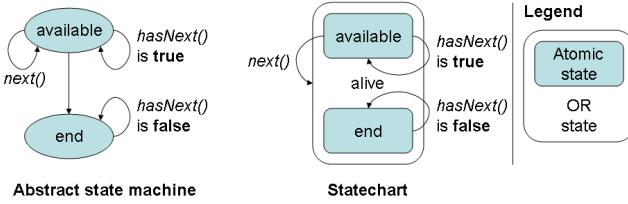


Figure 2. Read-only iterator state machine protocol

- Capture the usage protocol of Java iterators.
- Allow creating an arbitrary number of iterators over collections.
- Invalidate iterators before modification of the iterated collection.

2.2 State Machine Protocol

An iterator returns all elements of an underlying *collection* one by one. Collections in the Java standard library are lists or sets of objects. Their interface includes methods to add objects, remove objects, and test whether an object is part of the collection. The interface also defines a method `iterator` that creates a new iterator over the collection. Repeatedly calling `next` on an iterator returns each object contained in the iterated collection exactly once. The method `hasNext` determines whether another object is available or the iteration reached its end. It is illegal to call `next` once `hasNext` returns `false`. Figure 2 illustrates this protocol as a simple state machine.

Notice that `hasNext` is legal in both states but does not change state. We call `hasNext` a *dynamic state test*: its return value indicates what state the iterator is currently in. The next section will show how this protocol can be specified.

2.3 Iterator Interface Specification

States Through Refinement. We call the set of possible states of an object its *state space* and define it as part of the object’s interface. As suggested above, we can model the iterator state space with two states, `available` and `end`. In our approach, states are introduced by *refinement* of an existing state. State refinement corresponds to OR-states in Statecharts [20] and puts states into a tree hierarchy.

State refinement allows interfaces to, at the same time, *inherit* their supertypes’ state spaces, define additional (more fine-grained) states, and be properly *substitutable* as subtypes of extended interfaces [4]. Refinement guarantees that all new states defined in a subtype correspond to a state inherited from the supertype. States form a hierarchy rooted in a state `alive` defined in the root type `Object`. Iterators therefore define their state space as follows.

```
states available, end refine alive;
```

Typestates do *not* correspond to fields in a class. They describe an object’s state of execution abstractly and information about fields can be *tied* to typestates using state invariants (see section 3.1).

Access Permissions Capture Design Intent. Iterators have only two methods, but these have very different behavior. While `next` can *change* the iterator’s state, `hasNext` only *tests* the iterator’s state. And even when a call to `next` does not change the iterator’s state, it still advances the iterator to the next object in the sequence. `hasNext`, on the other hand, is *pure*: it does not modify the iterator at all.

We use a novel abstraction, *access permissions* (“permissions” for short), to capture this *design intent* as part of the iterator’s protocol. Permissions are associated with object references and govern how objects can be accessed through a given reference [7]. For `next` and `hasNext` we only need two kinds of permissions; more kinds of permissions will be introduced later.

- full permissions grant read/write access to the referenced object *and guarantee that no other reference has read/write access* to the same object.
- pure permissions grant read-only access to the referenced object *but assume that other permissions could modify the object*.

A distinguished full permission can co-exist with an arbitrary number of pure permissions to the same object. This property will be enforced when verifying protocol compliance. In a specification we write $\text{perm}(x)$ for a permission to an object referenced by x , where perm is one of the permission kinds. Access permissions carry state information about the referenced object. For example, “full(`this`) in `available`” represents a full permission for an object (`this`) that is in the `available` state.

Linear Logic Specifications. Methods can be specified with a *state transition* that describes how method parameters change state during method execution. We previously argued that existing typestate verification approaches are limited in their ability to express realistic state transitions [4] and proposed to capture method behavior more precisely with logical expressions.

Access permissions represent resources that have to be consumed upon usage—otherwise permissions could be freely duplicated, possibly violating other permissions’ assumptions. Therefore, we base our specifications on linear logic [18]. Pre- and post-conditions are separated with a linear implication (\multimap) and use conjunction (\otimes) and disjunction (\oplus).¹ In certain cases, internal choice (&, also called additive conjunction) has been useful [3]. These connectives represent the decidable multiplicative-additive fragment of linear logic (MALL).

Iterators illustrate that state transitions are often non-deterministic. For `next`, we can use an *imprecise* post-condition and specify `next` so that it requires a full permission in state `available` and returns the full permission in the

¹ “Tensor” (\otimes) corresponds to conjunction, “alternative” (\oplus) to disjunction, and “lollie” (\multimap) to implication in conventional logic. The key difference is that linear logic treats known facts as resources that are consumed when proving another fact. This fits well with our intuition of permissions as resources that give access to objects.

alive state. In a Statechart, this corresponds to transitioning to a state that contains substates (figure 2).

$\text{full}(\text{this}) \text{ in available} \multimap \text{full}(\text{this}) \text{ in alive}$

Dynamic state tests (like `hasNext`) require relating the (Boolean) method result to the state of the tested object (usually the receiver). A disjunction of conjunctions expresses the two possible outcomes of `hasNext` (figure 4) where each conjunction relates a possible method result to the corresponding receiver state. (We adopt the convention that \multimap binds weaker than \otimes and \oplus .)

$\text{pure}(\text{this}) \multimap (\text{result} = \text{true} \otimes \text{pure}(\text{this}) \text{ in available})$
 $\quad \oplus (\text{result} = \text{false} \otimes \text{pure}(\text{this}) \text{ in end})$

These specifications enforce the characteristic `hasNext / next` call pairing: `hasNext` determines the iterator's current state. If it returns true then it is legal to call `next`. The iterator is in an unknown state after `next` returns, and another `hasNext` call determines the iterator's new state.

2.4 Creating and Disposing Iterators

Multiple (independent) iterators are permitted for a single collection at the same time. However, the collection must not be modified while iteration is in progress. Standard implementations try to detect such situations of *concurrent modification* on a best-effort basis. But, ultimately, Java programmers have to make sure on their own that collections are not modified while iterated. (Note that “concurrent” modifications often occur in single-threaded programs [32].)

This section shows how the aliasing constraints between iterators and its collection can be handled. As we will see, this problem is largely orthogonal to specifying the relatively simple protocol for individual iterators that was discussed in the previous section.

Immutable Access Prevents Concurrent Modification. Access permissions can guarantee the absence of concurrent modification. The key observation is that when an iterator is created it stores a reference to the iterated collection in one of its fields. This reference should be associated with a permission that guarantees the collection's *immutability* while iteration is in progress. We include two previously proposed permissions [6] into our system in order to properly specify collections.

- immutable permissions grant read-only access to the referenced object *and guarantee that no reference has read/write access* to the same object.
- unique permissions grant read/write access *and guarantee that no other reference has* any access to the object.

Thus immutable permissions *cannot* co-exist with full permissions to the same object. We can specify the collection's `iterator` method using these permissions as follows. Notice how it *consumes* or *captures* the incoming receiver permission and returns an initial unique permission to a fresh

```
Collection c = new ...                                // legal
Iterator it = c.iterator();                          // legal
while(it.hasNext() && ...) {                         // legal
    Object o = it.next();                            // legal
    Iterator it2 = c.iterator();                      // legal
    while(it2.hasNext()) {                           // legal
        Object o2 = it2.next();                       // legal
        ...
    }
    if(it.hasNext() && c.size() == 3) {               // legal
        c.remove(it.next());                          // legal
        if(it.hasNext()) ... }                      // ILLEGAL
    Iterator it3 = c.iterator();                      // legal
```

Figure 3. A simple Iterator client

iterator object.

```
public class Collection {
    Iterator iterator() : immutable(this) -> unique(result)
}
```

It turns out that this specification precisely captures Sun's Java standard library implementation of iterators: Iterators are realized as inner classes that implicitly reference the collection they iterate.

Permission Splitting. How can we track permissions? Consider a client such as the one in figure 3. It gets a unique permission when first creating a collection. Then it creates an iterator which captures an immutable permission to the collection. However, the client later needs more immutable permissions to create additional iterators. Thus while a unique permission is intuitively stronger than an immutable permission we cannot just coerce the client's unique permission to an immutable permission and pass it to `iterator`: it would get captured by the newly created iterator, leaving the client with no permission to the collection at all.

In order to avoid this problem we use *permission splitting* in our verification approach. Before method calls we split the original permission into two, one of which is retained by the caller. Permissions are split so that their assumptions are not violated. In particular, we never duplicate a full or unique permission and make sure that no full permission co-exists with an immutable permission to the same object. Some of the legal splits are the following.

$$\begin{aligned} \text{unique}(x) &\Rightarrow \text{full}(x) \otimes \text{pure}(x) \\ \text{full}(x) &\Rightarrow \text{immutable}(x) \otimes \text{immutable}(x) \\ \text{immutable}(x) &\Rightarrow \text{immutable}(x) \otimes \text{immutable}(x) \\ \text{immutable}(x) &\Rightarrow \text{immutable}(x) \otimes \text{pure}(x) \end{aligned}$$

They allow the example client in figure 3 to retain an immutable permission when creating iterators, permitting multiple iterators and reading the collection directly at the same time.

Permission Joining Recovers Modifying Access. When splitting a full permission to a collection into immutable

```

interface Iterator<c:Collection, k:Fract> {
    states available, end refine alive

    boolean hasNext() :
        pure(this) —> (result = true ⊗ pure(this) in available)
            ⊕ (result = false ⊗ pure(this) in end)
    Object next() :
        full(this) in available —> full(this)
    void finalize() :
        unique(this) —> immutable(c, k)
}

interface Collection {
    void add(Object o) : full(this) —> full(this)
    int size() : pure(this) —> result ≥ 0 ⊗ pure(this)
    // remove(), contains() etc. similar

    Iterator<this, k> iterator() :
        immutable(this, k) —> unique(result)
}

```

Figure 4. Read-only Iterator and partial Collection interface specification

permissions we lose the ability to modify the collection. Intuitively, we would like to reverse permission splits to regain the ability to modify the collection.

Such *permission joining* can be allowed if we introduce the notion of fractions [6]. Essentially, fractions keep track of how often a permission was split. This later allows joining permissions (with known fractions) by putting together their fractions. A unique permission by definition holds a *full fraction* that is represented by one (1). We will capture fractions as part of our permissions and write $(perm)(x, k)$ for a given permission with fraction k . We usually do not care about the exact fraction and therefore implicitly quantify over all fractions. If a fraction does not change we often will omit it. Fractions allow us to define splitting and joining rules as follows.

$$\begin{aligned}
&\text{unique}(x, 1) \iff \text{full}(x, 1/2) \otimes \text{pure}(x, 1/2) \\
&\text{full}(x, k) \iff \text{immutable}(x, k/2) \otimes \text{immutable}(x, k/2) \\
&\text{immutable}(x, k) \iff \text{immutable}(x, k/2) \otimes \text{immutable}(x, k/2) \\
&\text{immutable}(x, k) \iff \text{immutable}(x, k/2) \otimes \text{pure}(x, k/2)
\end{aligned}$$

For example, we can split $\text{full}(it, 1/2)$ into $\text{full}(it, 1/4) \otimes \text{pure}(it, 1/4)$ and recombine them. Such reasoning lets our iterator client recover a unique iterator permission after each call into the iterator.

Recovering Collection Permissions. Iterators are created by trading a collection permission for a unique iterator permission. We essentially allow the opposite trade as well in order to modify a previously iterated collection again: We can safely consume a unique iterator permission and recover the permissions to its fields because no reference will be able to access the iterator anymore. A simple live variable analysis can identify when variables with unique permissions are no longer used. (As a side effect, a permission-based approach therefore allows identifying dead objects.)

```

Collection c = new ...           unique(c)
Iterator it<c, 1/2> = c.iterator();   immutable(c, 1/2) ⊗ unique(it)
while(it.hasNext() && ...) {
    immutable(c, 1/2) ⊗ unique(it) in available
    Object o = it.next();           immutable(c, 1/2) ⊗ unique(it)
    Iterator it2<c, 1/4> = c.iterator();   immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2)
    while(it2.hasNext()) {
        immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2) in available
        Object o2 = it2.next();       immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2)
        ...
    } // it2 dies
}                                     immutable(c, 1/2) ⊗ unique(it)
if(it.hasNext() && c.size() == 3) {
    immutable(c, 1/2) ⊗ unique(it) in available
    c.remove(it.next()); // it dies after next()
    unique(c) and no permission for it
    if(it.hasNext()) ...
}                                     // ILLEGAL
// it definitely dead
unique(c)
Iterator it3<c, 1/2> = c.iterator();   immutable(c, 1/2) ⊗ unique(it3)

```

Figure 5. Verifying a simple Iterator client

For lack of a more suitable location, we annotate the *finalize* method to indicate what happens when an iterator is no longer usable. And in order to re-establish *exactly* the permission that was originally passed to the iterator we parameterize *Iterator* objects with the collection permission’s fraction. The *finalize* specification can then release the captured collection permission from dead iterators. The complete specification for iterators and a partial collection specification are summarized in figure 4.

2.5 Client Verification

Figure 5 illustrates how our client from figure 3 can be verified by tracking permissions and splitting/joining them as necessary. After each line of code we show the current set of permissions on the right-hand side of the figure. We recover collection permissions from dead iterators as soon as possible. This lets us verify the entire example client. We correctly identify the seeded protocol violation.

2.6 Summary

We presented a specification of read-only iterators that prevents concurrent collection modification. To this end it associates collections and iterators with *access permissions*, defines a simple state machine to capture the iterator usage protocol, and tracks permission information using a decidable fragment of linear logic. Our logic-based specifications can relate objects to precisely specify method behavior in terms of typestates and support reasoning about dynamic tests.

3. Java Stream Implementations

I/O protocols are common examples for typestate-based protocol enforcement approaches [11, 12, 4]. This section sum-

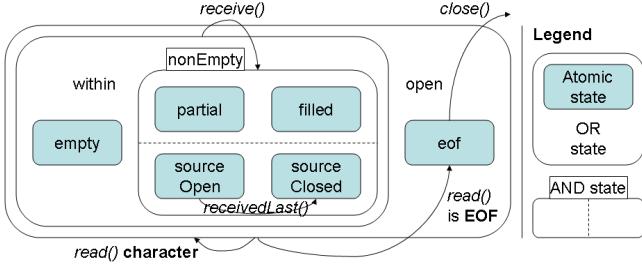


Figure 6. PipedInputStream’s state space (inside open)

marizes a case study in applying our approach to Java *character streams* and in particular *stream pipes* and *buffered input streams*. The section focuses on *implementation verification* of stream classes, which—to our knowledge—has not been attempted with typestates before. Implementation verification generalizes techniques shown in the previous section for client verification.

3.1 Stream Pipes

Pipes are commonly used in operating system shells to forward output from one process to another process. Pipes carry alphanumeric *characters* for a source to a sink. The Java I/O library includes a pair of classes, `PipedOutputStream` and `PipedInputStream`, that offers this functionality inside Java applications. This section provides a specification for Java pipes and shows how the classes implementing pipes in the Java standard library can be checked using our approach.

Informal Pipe Contract. In a nutshell, Java pipes work as follows: A character-producing “writer” writes characters into a `PipedOutputStream` (the “source”) that forwards them to a connected `PipedInputStream` (the “sink”) from which a “reader” can read them. The source forwards characters to the sink using the internal method `receive`. The writer calls `close` on the source when it is done, causing the source to call `receivedLast` on the sink (figure 7).

The sink caches received characters in a circular buffer. Calling `read` on the sink removes a character from the buffer (figure 8). Eventually the sink will indicate, using an *end of file token* (EOF, -1 in Java), that no more characters can be read. At this point the reader can safely close the sink. Closing the sink before EOF was read is unsafe because the writer may still be active.

The pipe classes in Sun’s standard library implementation have built-in runtime checks that throw exceptions in the following error cases: (1) closing the sink before the source, (2) writing to a closed source or pushing characters to the sink after the source was closed, and (3) reading from a closed sink. The specification we present here makes these error cases impossible.

State Space with Dimensions. The *source protocol* can be modeled with three states raw, open, and closed. raw indicates that the source is not connected to a sink yet. For technical reasons that are discussed below, we refine

open into ready and sending. The writer will always find the source in state ready.

For the *sink protocol* we again distinguish open and closed. A refinement of open helps capturing read’s protocol: The sink is within as long as read returns characters; the eof state is reached when read returns the EOF token. While within, we keep track of the sink’s buffer being empty or nonEmpty. We further refine nonEmpty into partial and filled, the latter corresponding to a full buffer.

At the same time, however, we would like to track whether the source was closed, i.e., whether `receivedLast` was called. We previously proposed *state dimensions* to address such separate concerns (here, the buffer filling and the source state) [4] with states that are independent from each other. State dimensions correspond to AND-states in Statecharts [20].

We can simply refine nonEmpty *twice, along different dimensions*. We call the states for the second dimension `sourceOpen` and `sourceClosed` with the obvious semantics. Note that we only need the additional source dimension while the buffer is nonEmpty; the source is by definition open (closed) in the empty (eof) state.² To better visualize the sink’s state space, figure 6 summarizes it as a Statechart.

Shared Modifying Access. Protocols for source and sink are formalized in figures 7 and 8 with specifications that work similar to the iterator example in the last section. However, the sink is conceptually modified through two distinct references, one held by the source and one held by the reader. In order to capture this, we introduce our last permission.

- share permissions grant read/write access to the referenced object but *assume that other permissions have read/write access as well*.

Conventional programming languages effectively always use share permissions for mutable state. Interestingly, share permissions are split and joined exactly like immutable permissions. Since share and immutable permissions cannot co-exist, our rules force a commitment to either one when initially splitting a full permission.

$$\begin{aligned} \text{full}(x, k) &\iff \text{share}(x, k/2) \otimes \text{share}(x, k/2) \\ \text{share}(x, k) &\iff \text{share}(x, k/2) \otimes \text{share}(x, k/2) \\ \text{share}(x, k) &\iff \text{share}(x, k/2) \otimes \text{pure}(x, k/2) \end{aligned}$$

State Guarantees. We notice that most modifying methods cannot change a stream’s state arbitrarily. For example, `read` and `receive` will never leave the open state and they cannot tolerate other permission to leave open.

We make this idea part of our access permissions. We include another parameter into permissions that specifies a *state guarantee*, i.e. a state that cannot be left even by modifying permissions. Thus a state guarantee (also called the permission’s *root*) corresponds to an “area” in a Statechart that cannot be left. As an example, we can write the permis-

²This is only *one way* of specifying the sink. It has the advantage that readers need not concern themselves with the internal communication between source and sink.

```

public class PipedOutputStream {
    states raw, open, closed refines alive;
    states ready, sending refines open;

    raw := sink = null
    ready := half(sink, open)
    sending := sink ≠ null
    closed := sink ≠ null

    private PipedInputStream sink;

    public PipedOutputStream() :
        1 → unique(this) in raw { }

    void connect(PipedInputStream snk) :
        full(this) in raw ⊗ half(snk, open) →
        full(this) in ready
    { sink = snk;                      store permission in field
        full(this) in open
    }

    public void write(int b) :
        full(this, open) in ready ⊗ b ≥ 0 → full(this, open) in ready
    { sink.receive(b);                  half(sink, open) from invariant
        returns half(sink, open)
        full(this, open) in ready
    }

    public void close() :
        full(this) in ready → full(this) in closed
    { sink.receivedLast();             half(sink, open) from invariant
        consumes half(sink, open)
        full(this) in closed
    }
}

```

Figure 7. Java PipedOutputStream (simplified)

sion needed for read as share(*this*, open). Without an explicit state guarantee, only alive is guaranteed (this is what we did for iterators).

State guarantees turn out to be crucial in making share and pure permissions useful because they guarantee a state even in the face of possible changes to the referenced object through other permissions. Moreover, if we combine them with state dimensions we get independent permissions for orthogonal object aspects that, e.g., let us elegantly model modifying iterators [3].

Explicit Fractions for Temporary Heap Sharing. When specifying the sink methods used by the source (receive and receivedLast) we have to ensure that the source can no longer call the sink after receivedLast so the sink can be safely closed. Moreover, in order to close the sink, we need to restore a permission rooted in alive. Thus the two share permissions for the sink have to be joined in such a way that there are definitely no other permissions relying on open (such permissions, e.g., could have been split off of one of the share permissions).

We extend the notion of fractions to accomplish this task. We use fractions to track, *for each state separately*, how many permissions rely on it. What we get is a *fraction function* that maps guaranteed states (i.e. the permission's

```

class PipedInputStream {
    stream = open, closed refines alive;
    position = within, eof refines open;
    buffer = empty, nonEmpty refines within;
    filling = partial, filled refines nonEmpty;
    source = sourceOpen, sourceClosed refines nonEmpty;

    empty := in ≤ 0 ⊖ closedByWriter = false
    partial := in ≥ 0 ⊖ in ≠ out
    filled := in = out
    sourceOpen := closedByWriter = false
    sourceClosed := closedByWriter ⊗ half(this, open)
    eof := in ≤ 0 ⊖ closedByWriter ⊗ half(this, open)

    private boolean closedByWriter = false;
    private volatile boolean closedByReader = false;
    private byte buffer[] = new byte[1024];
    private int in = -1, out = 0;

    public PipedInputStream(PipedOutputStream src) :
        full(src) in raw → half(this, open) ⊗ full(src) in open
    { unique(this) in open ⇒ half(this, open) ⊗ half(this, open)
        src.connect(this);           consumes one half(this, open)
    }                                half(this, open) ⊗ full(src) in open

    synchronized void receive(int b) :
        half(this, open) ⊗ b ≥ 0 → half(this, open) in nonEmpty
    { // standard implementation checks if pipe intact
        while(in == out)          half(this, open) in filled
            ... // wait a second
        half(this, open) in empty ⊕ partial
        if(in < 0) { in = 0; out = 0; }
        buffer[in++] (byte)(b & 0xFF);
        if(in >= buffer.length) in = 0;
    }                                half(this, open) in partial

    synchronized void receivedLast() :
        half(this, open) → 1
    { closedByWriter = true; }      this is now sourceClosed

    public synchronized int read() :
        share(this, open) → (result ≥ 0 ⊖ share(this, open))
        ⊕ (result = -1 ⊖ share(this, open) in eof)
    { ... } // analogous to receive()

    public synchronized void close() :
        half(this, open) in eof → unique(this) in closed
    { half(this, open) from eof invariant ⇒ unique(this, open)
        closedByReader = true;
        in = -1;
    }
}

```

Figure 8. Java PipedInputStream (simplified)

root and its super-states) to fractions. For example, if we split an initial unique permission for a PipedInputStream into two share permissions guaranteeing open then these permissions rely on open and alive with a 1/2 fraction each. (Iterator permissions root in alive and their fraction functions map alive to the given fraction.)

In order to close the sink, we have to make sure that there are *exactly* two share permissions relying on open. Fraction functions make this requirement precise. For readability, we use the abbreviation half in figure 8 that stands for the following permission.

$$\text{half}(x, \text{open}) \equiv \text{share}(x, \text{open}, \{\text{alive} \mapsto 1/2, \text{open} \mapsto 1/2\})$$

By adding fractions and moving the state guarantee up in the state hierarchy, the initial permission for the sink, unique(*this*, alive, {alive \mapsto 1}), can be regained from two half(*this*, open) permissions. half is the only permission with an explicit fraction function. All other specifications implicitly quantify over all fraction functions and leave them unchanged.

State Invariants Map Typestates to Fields. We now have a sufficient specification for both sides of the pipe. In order to verify their implementations we need to know what typestates correspond to in implementations. Our implementation verification extends Fugue’s approach of using *state invariants* to map states to predicates that describe the fields of an object in a given state [12]. We leverage our hierarchical state spaces and allow state invariants for states with refinements to capture invariants common to all substates of a state.

Figure 7 shows that the source’s state invariants describe its three states in the obvious way based on the field `snk` pointing to the sink. Notice that the invariant does not only talk about the sink’s state (as in Fugue) but uses permissions to control access through fields just as through local variables.

The sink’s state invariants are much more involved (figure 8) and define, e.g., what the difference between an empty buffer ($in < 0$) and a filled circular buffer ($in = out$) is. Interestingly, these invariants are all meticulously documented in the original Java standard library implementation for `PipedInputStream` [4]. The half permission to itself that the sink temporarily holds for the time between calls to `receivedLast` and `close` lets us verify that `close` is allowed to close the sink.

Verification with Invariants. Implementation checking assumes state invariants implied by incoming permissions and tracks changes to fields. Objects *have to be in a state whenever they yield control to another object*, including during method calls. For example, the source transitions to sending before calling the sink. However, the writer never finds the source in the sending state but always ready—sending never occurs in a method specification. We call states that are not observed by a client *intermediate states*. They help us deal with re-entrant calls (details in section 5.2). A practical syntax could make such intermediate states implicit.

Figures 7 and 8 show how implementation checking proceeds for most of the source’s and sink’s methods. We show in detail how field assignments change the sink’s state. The sink’s state information is frequently a disjunction of possible states. Dynamic tests essentially rule out states based on incompatible invariants. All of these tests are present in the

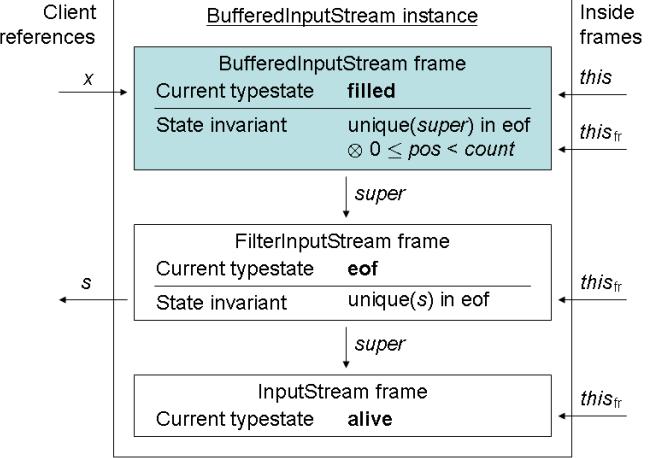


Figure 9. Frames of a `BufferedInputStream` instance in state `filled`. The blue *virtual frame* is in a different state than its super-frame.

original Java implementation; we removed additional non-null and state tests that are obviated by our approach. This not only shows how our approach forces necessary state tests but also suggests that our specifications could be used to *generate* such tests automatically.

3.2 Buffered Input Streams

A `BufferedInputStream` (or “buffer”, for short) wraps another “underlying” stream and provides buffering of characters for more efficient retrieval. We will use this example to illustrate our approach to handling inheritance. Compared to the original implementation, we made fields “private” in order to illustrate calls to overridden methods using `super`. We omit intermediate states in this specification.

Class Hierarchy. `BufferedInputStream` is a subclass of `FilterInputStream`, which in turn is a subclass of `InputStream`. `InputStream` is the abstract base class of all input streams and defines their protocol with informal documentation that we formalize in figure 10. It implements *convenience methods* such as `read(int[])` in terms of other—abstract—methods. `FilterInputStream` holds an underlying stream in a field `s` and simply forwards all calls to that stream (figure 10). `BufferedInputStream` overrides these methods to implement buffering.

Frames. The buffer occasionally calls overridden methods to read from the underlying stream. How can we reason about these internal calls? Our approach is based on Fugue’s *frames* for reasoning about inheritance [12]. Objects are broken into frames, one for each class in the object’s class hierarchy. A frame holds the fields defined in the corresponding class. We call the frame corresponding to the object’s runtime type the *virtual frame*, referred to with normal references (including `this`). Relative to a method, we call the current frame—corresponding to the class that the method is defined in—with `thisfr`, and the frame corresponding to

```

public abstract class InputStream {
    states open, closed refine alive;
    states within, eof refine open;

    public abstract int read() :
        share(thisfr, open) —> (result ≥ 0 ⊗ share(thisfr, open))
            ⊕ (result = -1 ⊗ share(thisfr, open) in eof)
    public abstract void close() :
        full(thisfr, alive) in open —> full(thisfr, alive) in closed

    public int read(byte[] buf) :
        share(this, open) ⊗ buf ≠ null —>
            (result = -1 ⊗ share(this, open) in eof) ⊕
            (result ≥ 0 ⊗ share(this, open))
    { ... for(...)

        ... int c = this.read() ...
    }

    public class FilterInputStream extends InputStream {
        within := unique(s) in within
        eof := unique(s) in eof
        closed := unique(s) in closed

        private volatile InputStream s;

        protected FilterInputStream(InputStream s)
            unique(s, alive) in open —> unique(thisfr, alive) in open
        { this.s = s; }
        ... // read() and close() forward to s
    }
}

```

Figure 10. Java FilterInputStream forwards all calls to underlying InputStream (simplified)

the immediate superclass is called *super* frame. Figure 9 shows a sample BufferedInputStream instance with its three frames.

Frame Permissions. In our approach, a permission actually grants access to a particular frame. The permissions we have seen so far give a client access to the referenced object’s virtual frame. Permissions for other frames are only accessible from inside a subclass through *super*.

Figure 9 illustrates that a BufferedInputStream’s state can differ from the state its filter frame is in: the filter’s state might be eof (when the underlying stream reaches eof) while the buffer’s is still within (because the buffer array still holds unread characters). The state invariants in figure 11 formalize this. They let us verify that *super* calls in the buffer implementation respect the filter’s protocol.

Because the states of frames can differ it is important to enforce that a permission is only ever used to access fields in the frame it grants permission to. In specifications we specifically mark permissions that will actually access fields (and not just call other methods) of the receiver with *this_{fr}*. We require all methods that use these permissions to be overridden. On the other hand, convenience methods such as *read(int[])* can operate with permissions to the virtual frame and need not be overridden (figure 10).

```

public class BufferedInputStream
    extends FilterInputStream {
    states depleted, filled refine within;

    closed := unique(super) in closed ⊗ buf = null
    open := unique(buf)
    filled := pos < count ⊗ unique(super) in open
    depleted := pos ≥ count ⊗ unique(super) in within
    eof := pos ≥ count ⊗ unique(super) in in eof

    private byte buf[] = new byte[8192];
    private int count = 0, pos = 0;

    public BufferedInputStream(InputStream s)
        unique(s) in open —> unique(thisfr) in open
    { count = pos = 0 ⊗ unique(buf)
        super(s); unique(super) in open
    }

    public synchronized int read() {
        if(pos >= count)
            share(thisfr, open) in depleted ⊕ eof
            fill(); share(thisfr, open) in filled ⊕ eof
        if(pos >= count)
            return -1; returns share(thisfr, open) in eof
        any path: share(thisfr, open) in filled
        return buf[pos++] & 0xFF;
    } share(thisfr, open) in filled ⊕ eof

    private void fill()
        share(thisfr, open) in depleted ⊕ eof —>
            share(thisfr, open) in filled ⊕ eof
    { invariant: unique(super) in within ⊕ eof
        count = pos = 0; note: assumes buffer was fully read
        int b = super.read(); unique(super) in within ⊕ eof
        while(b >= 0) {
            unique(super) in within
            buf[count++] = (byte) (b & 0xFF);
            share(thisfr, open) in filled
            if(count >= buf.length) break;
            b = super.read(); unique(super) in within ⊕ eof
        } if loop never taken, share(thisfr, open) in eof
    } share(this, open) in filled ⊕ eof

    public synchronized void close() {
        buf = null; invariant: unique(super) in open
        super.close(); unique(super) in closed
    } full(thisfr, alive) in closed
}

```

Figure 11. BufferedInputStream caches characters from FilterInputStream base class

This distinction implies that *fill* (figure 11) *cannot* call *read(int[])* (because it does not have a suitable virtual frame permission) but *only* *super.read()*. This is imperative for the correctness of *fill* because a dynamically dispatched call would lead back into the—still empty—buffer, causing an infinite loop. (One can trigger exactly this effect in the Java 6 implementation of BufferedInputStream.)

3.3 Summary

This section showed how our approach can be used to verify realistic Java pipe and buffered input stream implementations. The notion of access permissions is central to our approach. Overall, we introduced five different kinds of permissions (figure 1). While three kinds are adapted from existing work [7, 12] we recently proposed full and pure permissions [3]. State guarantees and temporary state information increase the usefulness of “weak” (share and pure) permissions. Permission splitting and joining is flexible enough to model temporary aliasing on the stack (during method calls) and in the heap (e.g., in pipes and iterators). Permission-based state invariants enable reasoning about protocol implementations. We handle inheritance based on frames [12] and permit dynamic dispatch within objects for convenience methods.

4. Formal Language

This section formalizes an object-oriented language with protocol specifications. We briefly introduce expression and class declaration syntax before defining state spaces, access permissions, and permission-based specifications. Finally, we discuss handling of inheritance and enforcement of behavioral subtyping.

4.1 Syntax

Figure 12 shows the syntax of a simple class-based object-oriented language. The language is inspired by Featherweight Java (FJ, [24]); we will extend it to include type-state protocols in the following subsections. We identify classes (C), methods (m), and fields (f) with their names. As usual, x ranges over variables including the distinguished variable *this* for the receiver object. We use an overbar notation to abbreviate a list of elements. For example, $\overline{x : T} = x_1:T_1, \dots, x_n:T_n$. Types (T) in our system include Booleans (`bool`) and classes.

Programs are defined with a list of class declarations and a main expression. A class declaration CL gives the class a unique name C and defines its fields, methods, typestates, and state invariants. A constructor is implicitly defined with the class’s own and inherited fields. Fields (F) are declared with their name and type. Each field is mapped into a part of the state space n that can depend on the field (details in section 5.2). A method (M) declares its result type, formal parameters, specification and a body expression. State refinements R will be explained in the next section; method specifications MS and state invariants N are deferred to section 4.4.

We syntactically distinguish pure terms t and possibly effectful expressions e . Arguments to method calls and object construction are restricted to terms. This simplifies reasoning about effects [30, 9] by making execution order explicit.

Notice that we syntactically restricts field access and assignments to fields of the receiver class. Explicit “getter” and “setter” methods can be defined to give other objects access to fields. Assignments evaluate to the *previous* field value.

<i>programs</i>	$PR ::= \langle \overline{CL}, e \rangle$
<i>class decl.</i>	$CL ::= \text{class } C \text{ extends } C' \{ \overline{F} \overline{R} \overline{I} \overline{N} \overline{M} \}$
<i>field decl.</i>	$F ::= f : T \text{ in } n$
<i>meth. decl.</i>	$M ::= T m(\overline{T} \overline{x}) : MS = e$
<i>state decl.</i>	$R ::= d = \overline{s} \text{ refines } s_0$
<i>terms</i>	$t ::= x \mid o \mid \text{true} \mid \text{false}$ $\mid t_1 \text{ and } t_2 \mid t_1 \text{ or } t_2 \mid \text{not } t$
<i>expressions</i>	$e ::= t \mid f \mid \text{assign } f := t$ $\mid \text{new } C(\overline{t}) \mid t_0.m(\overline{t}) \mid \text{super}.m(\overline{t})$ $\mid \text{if}(t, e_1, e_2) \mid \text{let } x = e_1 \text{ in } e_2$
<i>values</i>	$v ::= o \mid \text{true} \mid \text{false}$
<i>references</i>	$r ::= x \mid f \mid o$
<i>types</i>	$T ::= C \mid \text{bool}$
<i>nodes</i>	$n ::= s \mid d$
<i>classes</i>	C
<i>fields</i>	f
<i>variables</i>	x
<i>objects</i>	o
<i>methods</i>	m
<i>states</i>	s
<i>dimensions</i>	d

Figure 12. Core language syntax. Specifications (I, N, MS) in figure 14.

4.2 State Spaces

State spaces are formally defined as a list of state refinements (see figure 12). A state refinement (R) refines an existing state in a new dimension with a set of mutually exclusive sub-states. We use s and d to range over state and dimension names, respectively. A *node* n in a state space can be a state or dimension. State refinements are inherited by subclasses. We assume a root state *alive* that is defined in the root class *Object*.

We define a variety of helper judgments for state spaces in figure 13. $\text{refinements}(C)$ determines the list of state refinements available in class C . $C \vdash A \text{ wf}$ defines well-formed state assumptions. Assumptions A combine states and are defined in figure 14. Conjunctive assumptions have to cover orthogonal parts of the state space. $C \vdash n \leq n'$ defines the substate relation for a class. $C \vdash A \# A'$ defines orthogonality of state assumptions. A and A' are orthogonal if they refer to different (orthogonal) state dimensions. $C \vdash A \prec n$ defines that a state assumption A only refers to states underneath a root node n . $C \vdash A \ll n$ finds the tightest such n .

4.3 Access Permissions

Access permissions p give references permission to access an object. Permissions to objects are written $\text{access}(r, n, g, k, A)$ (figure 14). (We wrote $\text{perm}(r, n, g)$ in A before.) The additional parameter k allows us to uniformly represent all permissions as explained below.

- Permissions are granted to references r . References can be variables, locations, and fields.
- Permissions apply to a particular *subtree* in the space of r that is identified by its root node n . It rep-

$$\begin{array}{c}
\frac{\text{refinements}(\text{Object}) = \cdot \quad \text{class } C \text{ extends } C' \{ \overline{F} \overline{R} \dots \} \quad \text{refinements}(C') = \overline{R'}}{\text{refinements}(C) = \overline{R'}, \overline{R}} \quad \frac{}{n \text{ in refinements}(C)} \\
\frac{C \vdash A_1 \text{ wf} \quad C \vdash A_2 \text{ wf} \quad C \vdash A_1 \text{ wf} \quad A_1 \# A_2 \quad C \vdash A_2 \text{ wf}}{C \vdash A_1 \oplus A_2 \text{ wf}} \quad \frac{d = \overline{s} \text{ refines } s \in \text{refinements}(C) \quad C \vdash d \leq s}{C \vdash s_i \leq d} \quad \frac{C \vdash n \text{ wf}}{C \vdash n \leq n} \\
\frac{C \vdash n \leq n'' \quad C \vdash n'' \leq n'}{C \vdash n \leq n'} \quad \frac{d = \overline{s} \text{ refines } s^* \in \text{refinements}(C) \quad d' = \overline{s'} \text{ refines } s^* \in \text{refinements}(C) \quad d \neq d'}{C \vdash d \# d'} \\
\frac{C \vdash n_1 \leq n'_1 \quad C \vdash n'_1 \# n'_2 \quad C \vdash n_2 \leq n'_2 \quad C \vdash A' \# A}{C \vdash n_1 \# n_2} \quad \frac{C \vdash A \# A'}{C \vdash A \# A'} \quad \frac{C \vdash A_{1,2} \# A}{C \vdash A_1 \otimes A_2 \# A} \quad \frac{C \vdash A_{1,2} \# A}{C \vdash A_1 \oplus A_2 \# A} \quad \frac{C \vdash n' \leq n}{C \vdash n' \prec n} \\
\frac{C \vdash A_{1,2} \prec n \quad C \vdash A_1 \otimes A_2 \text{ wf}}{C \vdash A_1 \otimes A_2 \prec n} \quad \frac{C \vdash A_{1,2} \prec n \quad C \vdash A_1 \oplus A_2 \text{ wf}}{C \vdash A_1 \oplus A_2 \prec n} \quad \frac{C \vdash A \prec n \quad \forall n' : C \vdash A \prec n' \text{ implies } n \leq n'}{C \vdash A \ll n}
\end{array}$$

Figure 13. State space judgments (assumptions A defined in figure 14)

resents a *state guarantee* (section 3). Other parts of the state space are unaffected by the permission.

- The *fraction function* g tracks for each node on the path from n to alive a symbolic fraction [6]. The fraction function keeps track of how often permissions were split at different nodes in the state space so they can be coalesced later (see section 5.5).
- The *subtree fraction* k encodes the level of access granted by the permission. $k > 0$ grants modifying access. $k < 1$ implies that other potentially modifying permissions exist. Fraction variables z are conservatively treated as a value between 0 and 1, i.e., $0 < z < 1$.
- An *state assumption* A expresses state knowledge within the permission’s subtree. Only full permissions can permanently make state assumptions until they modify the object’s state themselves. For weak permissions, the state assumption is *temporary*, i.e. lost after any effectful expression (because the object’s state may change without the knowledge of r).

We can encode unique, full, share, and pure permissions as follows. In our formal treatment we omit immutable permissions, but it is straightforward to encode them with an additional “bit” that distinguishes immutable and share permissions.

$$\begin{aligned}
\text{unique}(r, n, g) \text{ in } A &\equiv \text{access}(r, n, \{g, n \mapsto 1\}, 1, A) \\
\text{full}(r, n, g) \text{ in } A &\equiv \text{access}(r, n, g, 1, A) \\
\text{share}(r, n, g, k) \text{ in } A &\equiv \text{access}(r, n, g, k, A) \quad (0 < k < 1) \\
\text{pure}(n, n, g) \text{ in } A &\equiv \text{access}(r, n, g, 0, A)
\end{aligned}$$

4.4 Permission-Based Specifications

We combine atomic permissions (p) and facts about Boolean values (q) using linear logic connectives (figure 14). We also include existential ($\exists z : H.P$) and universal quantification of fractions ($\forall z : H.P$) to alleviate programmers from writing concrete fraction functions in most cases. We type all expressions as an existential type (E).

$$\begin{array}{ll}
\text{permissions} & p ::= \text{access}(r, n, g, k, A) \\
\text{facts} & q ::= t = \text{true} \mid t = \text{false} \\
\text{assumptions} & A ::= n \mid A_1 \otimes A_2 \mid A_1 \oplus A_2 \\
\text{fraction fct.} & g ::= z \mid \overline{n \mapsto v} \\
& \quad \mid g/2 \mid g_1, g_2 \\
\text{fractions} & k ::= 1 \mid 0 \mid z \mid k/2 \\
\text{predicates} & P ::= p \mid q \\
& \quad \mid P_1 \otimes P_2 \mid \mathbf{1} \\
& \quad \mid P_1 \& P_2 \mid \top \\
& \quad \mid P_1 \oplus P_2 \mid \mathbf{0} \\
& \quad \mid \exists z : H.P \mid \forall z : H.P \\
\text{method specs} & MS ::= P \multimap E \\
\text{expr. types} & E ::= \exists x : T.P \\
\text{state inv.} & N ::= n = P \\
\text{initial state} & I ::= \text{initially } \langle \exists f : T.P, S \rangle \\
\text{precise state} & S ::= s_1 \otimes \dots \otimes s_n \\
\text{fract. terms} & h ::= g \mid k \\
\text{fract. types} & H ::= \text{Fract} \mid \overline{n} \rightarrow \text{Fract} \\
\text{fract. vars.} & z
\end{array}$$

Figure 14. Permission-based specifications

Method specifications. Methods are specified with a linear implication (\multimap) of predicates (MS). The left-hand side of the implication (method pre-condition) may refer to method receiver and formal parameters. The right-hand side (post-condition) existentially quantifies the method result (a similar technique is used in Vault [11]). We refer to the receiver with *this* and usually call the return value *result*.

State invariants. We decided to use linear logic predicates for state invariants as well (N). In general, several of the defined state invariants will have to be satisfied at the same time. This is due to our hierarchical state spaces. Each class declares an initialization predicate and a start state (I) that are used for object construction (instead of an explicit constructor).

4.5 Handling Inheritance

Permissions give access to a particular frame, usually the virtual frame (see section 3.2) of an object. Permissions to the virtual frame are called *object permissions*. Because of subtyping, the precise frame referenced by an object permission is statically unknown.

$$\text{references } r ::= \dots \mid \text{super} \mid \text{this}_{\text{fr}}$$

In order to handle inheritance, we distinguish references to the receiver’s “current” frame (this_{fr}) and its super-frame (super). Permissions for these “special” references are called *frame permissions*. A this_{fr} permission grants access to fields and can be used in method specifications. Permissions for super are needed for super-calls and are only available in state invariants. All methods requiring a this_{fr} permission must be overridden because such methods rely on being defined in a particular frame to access its fields.

4.6 Behavioral Subtyping

Subclasses should be allowed to define their own specifications, e.g. to add precision or support additional behavior [4]. However, subclasses need to be *behavioral subtypes* [29] of the extended class. Our system enforces behavioral subtyping in two steps. Firstly, state space inheritance conveniently guarantees that states of subclasses *always* correspond to states defined in superclasses [4]. Secondly, we make sure that every overriding method’s specification implies the overridden method’s specification [4] using the override judgment (figure 16) that is used in checking method declarations. This check leads to method specifications that are contra-variant in the domain and co-variant in the range as required by behavioral subtyping.

5. Modular Typestate Verification

This section describes a static modular typestate checking technique for access permissions similar to conventional typechecking. It guarantees at compile-time that protocol specifications will never be violated at runtime. We emphasize that our approach does not require tracking typestates at run time.

A companion technical report contains additional judgments and a soundness proof for a fragment of the system presented in this paper [5]. The fragment does not include inheritance and only supports permissions for objects as a whole. State dimensions are omitted and specifications are deterministic. The fragment does include full, share, and pure permissions with fractions and temporary state information.

5.1 Permission Tracking

We permission-check an expression e with the judgment $\Gamma; \Delta \vdash_C^i e : \exists x : T.P \setminus \mathcal{E}$. This is read as, “in valid context Γ and linear context Δ , an expression e executed within receiver class C has type T , yields permissions P , and affects fields \mathcal{E} ”. Permissions Δ are consumed in the process. We omit the receiver C where it is not required for

checking a particular syntactic form. The set \mathcal{E} keeps track of fields that were assigned to, which is important for the correct handling of permissions to fields. It is omitted when empty. The marker i in the judgment can be 0 or 1 where $i = 1$ indicates that states of objects in the context may change during evaluation of the expression. This will help us reason about temporary state assumptions. A combination of markers with $i \vee j$ is 1 if at least one of the markers is 1.

$$\begin{array}{lll} \text{valid contexts} & \Gamma ::= \cdot \mid \Gamma, x : T \mid \Gamma, z : H \mid \Gamma, q \\ \text{linear contexts} & \Delta ::= \cdot \mid \Delta, P \\ \text{effects} & \mathcal{E} ::= \cdot \mid \mathcal{E}, f \end{array}$$

Valid and linear contexts distinguish valid (permanent) information (Γ) from resources (Δ). Resources are tracked linearly, forbidding their duplication, while facts can be used arbitrarily often. (In logical terms, contraction is defined for facts only). The valid context types object variables, fraction variables, and location types and keeps track of facts about terms q . Fraction variables are tracked in order to handle fraction quantification correctly. The linear context holds currently available resource predicates.

The judgment $\Gamma \vdash t : T$ types terms. It includes the usual rule for subsumption based on nominal subtyping induced by the extends relation (figure 16). Term typing is completely standard and can be found in the companion report. The companion report also includes rules for formally typing fractions and fraction functions [5].

Our expression checking rules are syntax-directed up to reasoning about permissions. Permission reasoning is deferred to a separate judgment $\Gamma; \Delta \vdash P$ that uses the rules of linear logic to prove the availability of permissions P in a given context. This judgment will be discussed in section 5.5. Permission checking rules for most expressions appear in figure 15 and are described in turn. Packing, method calls, and field assignment are discussed in following subsections. Helper judgments are summarized in figure 16. The notation $[e'/x]e$ substitutes e' for occurrences of x in e .

- P-TERM embeds terms. It formalizes the standard logical judgment for existential introduction and has no effect on existing objects.
- P-FIELD checks field accesses analogously.
- P-NEW checks object construction. The parameters passed to the constructor have to satisfy initialization predicate P and become the object’s initial field values. The new existentially quantified object is associated with a unique permission to the root state that makes state assumptions according to the declared start state A . Object construction has no effect on existing objects.

The judgment init (figure 16) looks up initialization predicate and start state for a class. The start state is a conjunction of states (figure 14). The initialization predicate is the invariant needed for the start state.

- P-IF introduces non-determinism into the system, reflected by the disjunction in its type. We make sure that the predicate is of Boolean type and then assume its truth

$$\begin{array}{c}
\frac{\Gamma \vdash t : T \quad \Gamma; \Delta \vdash [t/x]P}{\Gamma; \Delta \vdash^0 t : \exists x : T.P} \text{ P-TERM} \quad \frac{\text{localFields}(C) = \overline{f : T} \quad \Gamma; \Delta \vdash [f_i/x]P}{\Gamma; \Delta \vdash^0_C f_i : \exists x : T_i.P} \text{ P-FIELD} \\
\\
\frac{\Gamma \vdash \overline{t : T} \quad \text{init}(C) = \langle \exists \overline{f : T}.P, A \rangle \quad \Gamma; \Delta \vdash [\overline{t/f}]P}{\Gamma; \Delta \vdash^0 \text{new } C(\overline{t}) : \exists x : C.\text{access}(x, \text{alive}, \{\text{alive} \mapsto 1\}, 1, A)} \text{ P-NEW} \\
\\
\frac{\Gamma \vdash t : \text{bool} \quad (\Gamma, t = \text{true}); \Delta \vdash^i e_1 : \exists x : T.P_1 \setminus \mathcal{E}_1 \quad (\Gamma, t = \text{false}); \Delta \vdash^j e_2 : \exists x : T.P_2 \setminus \mathcal{E}_2}{\Gamma; \Delta \vdash^{i \vee j} \text{if}(t, e_1, e_2) : \exists x : T.P_1 \oplus P_2 \setminus \mathcal{E}_1 \cup \mathcal{E}_2} \text{ P-IF} \\
\\
\frac{\Gamma; \Delta \vdash^i e_1 : \exists x : T.P \setminus \mathcal{E}_1 \quad (\Gamma, x : T); (\Delta', P) \vdash^j e_2 : E_2 \setminus \mathcal{E}_2 \quad i = 1 \text{ implies no temporary assumptions in } \Delta' \quad \text{Fields in } \mathcal{E}_1 \text{ do not occur in } \Delta'}{\Gamma; (\Delta, \Delta') \vdash^{i \vee j} \text{let } x = e_1 \text{ in } e_2 : E_2 \setminus \mathcal{E}_1 \cup \mathcal{E}_2} \text{ P-LET} \\
\\
\frac{(\overline{x : T}, \text{this} : C); P \vdash^i_C e : \exists result : T_r.P_r \otimes \top \setminus \mathcal{E} \quad E = \exists result : T_r.P_r \quad \text{override}(m, C, \forall \overline{x : T}.P \multimap E)}{T_r m(\overline{T} \overline{x}) : P \multimap E = e \text{ ok in } C} \text{ P-METH} \\
\\
\frac{\dots \quad \overline{M} \text{ ok in } C \quad \overline{M} \text{ overrides all methods with } \text{this}_{fr} \text{ permissions in } C'}{\text{class } C \text{ extends } C' \{ \overline{F} \overline{R} \overline{I} \overline{N} \overline{M} \} \text{ ok}} \text{ P-CLASS} \quad \frac{\overline{CL} \text{ ok} \quad \cdot ; \cdot \vdash^i e : E \setminus \mathcal{E}}{\langle \overline{CL}, e \rangle : E} \text{ P-PROG}
\end{array}$$

Figure 15. Permission checking for expressions (part 1) and declarations

$$\begin{array}{c}
\frac{\text{class } C \text{ extends } C' \{ \dots \} \in \overline{CL}}{C \text{ extends } C'} \quad \frac{\text{class } C \{ \dots \overline{M} \dots \} \in \overline{CL} \quad T_r m(\overline{T} \overline{x}) : P \multimap \exists result : T_r.P' = e \in \overline{M}}{\text{mtype}(m, C) = \forall \overline{x : T}.P \multimap \exists result : T_r.P'} \\
\\
\frac{C \text{ extends } C' \quad \text{mtype}(m, C') = \forall \overline{x : T}.MS' \text{ implies } (\overline{x : T}, \text{this} : C); \cdot \vdash MS \multimap MS'}{\text{override}(m, C, \forall \overline{x : T}.MS)} \quad \frac{\text{class } C \dots \{ \overline{F} \dots \} \in \overline{CL}}{\text{localFields}(C) = \overline{F}}
\end{array}$$

$$\frac{\text{init}(\text{Object}) = (\mathbf{1}, \text{alive})}{\text{init}(C) = \langle \exists \overline{f' : T'}, \overline{f : T}.P' \otimes P, A \rangle}$$

$$\frac{\text{pred}_C(n) = P \quad P = \bigotimes_{n' \leq n'' < n} \text{pred}_C(n'')}{\text{pred}_C(n', n) = P} \quad \frac{\text{inv}_C(A) = P \Rightarrow n'}{\text{inv}_C(n, A) = P \otimes \text{pred}_C(n', n) \otimes \text{pred}_C(n)}$$

$$\frac{\text{inv}_C(A_i) = P_i \Rightarrow n_i \quad \text{pred}_C(n_i, n) = P'_i \quad n_1 \otimes n_2 \ll n \quad (i = 1, 2)}{\text{inv}_C(A_1 \otimes A_2) = P_1 \otimes P'_1 \otimes P_2 \otimes P'_2 \Rightarrow n}$$

$$\frac{\text{inv}_C(A_i) = P_i \Rightarrow n_i \quad \text{pred}_C(n_i, n) = P'_i \quad n_1 \oplus n_2 \ll n \quad (i = 1, 2)}{\text{inv}_C(A_1 \oplus A_2) = (P_1 \otimes P'_1) \oplus (P_2 \otimes \text{pred}_C(n_2, n)) \Rightarrow n}$$

$$\frac{\text{only pure permissions in } P}{\text{effectsAllowed}(P) = 0} \quad \frac{\text{exists share or full permission in } P}{\text{effectsAllowed}(P) = 1}$$

Figure 16. Protocol verification helper judgments

(falsehood) in checking the *then* (*else*) branch. This approach lets branches make use of the tested condition.

- P-LET checks a let binding. The linear context used in checking the second subexpression must not mention

fields affected by the first expression. This makes sure that outdated field permissions do not “survive” assignments or packing. Moreover, temporary state information is dropped if the first subexpression has side effects.

A program consists of a list of classes and a main expression (P-PROG, figure 15). As usual, the class table CL is globally available. The main expression is checked with initially empty contexts. The judgment $CL \text{ ok}$ (P-CLASS) checks a class declaration. It checks fields, states, and invariants for syntactic correctness (omitted here) and verifies consistency between method specifications and implementations using the judgment $M \text{ ok}$ in C . P-METH assumes the specified pre-condition of a method (i.e. the left-hand side of the linear implication) and verifies that the method’s body expression produces the declared post-condition (i.e. the right-hand side of the implication). Conjunction with \top drops excess permissions, e.g., to dead objects. The override judgment concisely enforces behavioral subtyping (see section 4.6). A method itself is not a linear resource since all resources it uses (including the receiver) are passed in upon invocation.

5.2 Packing and Unpacking

We use a refined notion of *unpacking* [12] to gain access to fields: we unpack and pack a specific permission. The access we gain reflects the permission we unpacked. Full and shared permissions give modifying access, while a pure permission gives read-only access to underlying fields.

To avoid inconsistencies, objects are always fully packed when methods are called. To simplify the situation, only one permission can be unpacked at the same time. Intuitively, we “focus” [13] on that permission. This lets us unpack share like full permissions, gaining full rather than shared access to underlying fields (if available). The syntax for packing and unpacking is as follows.

$$\begin{array}{lcl} \text{expressions } e ::= \dots & | & \text{unpack}(n, k, A) \text{ in } e \\ & | & \text{pack to } A \text{ in } e \end{array}$$

Packing and unpacking always affects the receiver of the currently executed method. The unpack parameters express the programmer’s expectations about the permission being unpacked. For simplicity, an explicit subtree fraction k is part of unpack expressions. It could be inferred from a programmer-provided permission kind, e.g. share.

Typechecking. In order for pack to work properly we have to “remember” the permission we unpacked. Therefore we introduce unpacked as an additional linear predicate.

$$\begin{array}{lcl} \text{permissions } p ::= \dots & | & \text{unpacked}(n, g, k, A) \end{array}$$

The checking rules for packing and unpacking are given in figure 18. Notice that packing and unpacking always affects permissions to $this_{fr}$. (We ignore substitution of $this$ with an object location at runtime here.)

P-UNPACK first derives the permission to be unpacked. The judgment inv determines a predicate for the receiver’s fields based on the permission being unpacked. It is used when checking the body expression. An unpacked predicate is added into the linear context. We can prevent multiple permissions from being unpacked at the same time using a straightforward dataflow analysis (omitted here).

$$\begin{aligned} \text{inv}_C(n, g, k, A) &= \text{inv}_C(n, A) \otimes \text{purify}(\text{above}_C(n)) \\ \text{inv}_C(n, g, 0, A) &= \text{purify}(\text{inv}_C(n, A) \otimes \text{above}_C(n)) \\ \text{where } \text{above}_C(n) &= \bigotimes_{n':n < n' \leq \text{alive}} \text{pred}_C(n') \end{aligned}$$

Figure 17. Invariant construction (purify in figure 19)

P-PACK does the opposite of P-UNPACK. It derives the predicate necessary for packing the unpacked permission and then assumes that permission in checking the body expression. The new state assumption A can differ from before only if a modifying permission was unpacked. Finally, the rule ensures that permissions to fields do not “survive” packing.

Invariant transformation. The judgment $\text{inv}_C(n, g, k, A)$ determines what permissions to fields are implied by a permission access $(this_{fr}, n, g, k, A)$ for a frame of class C . It is defined in figure 17 and uses a purify function (figure 19) to convert arbitrary into pure permissions.

Unpacking a full or shared permission with root node n yields purified permissions for nodes “above” n and includes invariants following from state assumptions as-is. Conversely, unpacking a pure permission yields completely purified permissions.

5.3 Calling Methods

Checking a method call involves proving that the method’s pre-condition is satisfied. The call can then be typed with the method’s post-condition.

Unfortunately, calling a method can result into reentrant callbacks. In order to ensure that objects are consistent when called we require them to be fully packed before method calls. This reflects that aliased objects always have to be prepared for reentrant callbacks.

This rule is not a limitation because we can always pack to some intermediate state although it may be inconvenient in practice. Notice that such *intermediate packing* obviates the need for adoption while allowing focus [13]: the intermediate state represents the situation where an adopted object was taken out of the adopting object. Inferring intermediate states as well as identifying where reentrant calls are impossible (intermediate packing avoidance) are important areas for future research.

Virtual calls. Virtual calls are dynamically dispatched (rule P-CALL). In virtual calls, frame and object permissions are identical because object permissions simply refer to the object’s virtual frame. This is achieved by substituting the given receiver for both $this$ and $this_{fr}$.

Super calls. Super calls are statically dispatched (rule P-SUPER). Recall that *super* is used to identify permissions to the super-frame. We substitute *super* only for $this_{fr}$. We omit a substitution of $this$ for the receiver ($this$ again) for clarity.

$$\begin{array}{c}
\frac{\Gamma; \Delta \vdash_C \text{access}(\text{this}_{\text{fr}}, n, g, k, A) \quad \text{receiver packed} \\
k = 0 \text{ implies } i = 0 \quad \Gamma; (\Delta', \text{inv}_C(n, g, k, A), \text{unpacked}(n, g, k, A)) \vdash_C^i e : E \setminus \mathcal{E}}{\Gamma; (\Delta, \Delta') \vdash_C^i \text{unpack}(n, k, A) \text{ in } e : E \setminus \mathcal{E}} \text{ P-UNPACK} \\
\\
\frac{\Gamma; \Delta \vdash_C \text{inv}_C(n, g, k, A) \otimes \text{unpacked}(n, g, k, A') \quad k = 0 \text{ implies } A = A' \\
\Gamma; (\Delta', \text{access}(\text{this}_{\text{fr}}, n, g, k, A)) \vdash_C^i e : E \setminus \mathcal{E} \quad \text{localFields}(C) = \overline{f : T \text{ in } n \quad \text{Fields do not occur in } \Delta'}}{\Gamma; (\Delta, \Delta') \vdash_C^i \text{pack } n \text{ to } A \text{ in } e : E \setminus \overline{f}} \text{ P-PACK} \\
\\
\frac{\Gamma \vdash t_0 : C_0 \quad \Gamma \vdash \overline{t : T} \quad \Gamma; \Delta \vdash [t_0/\text{this}][t_0/\text{this}_{\text{fr}}]\overline{[t/\bar{x}]P} \\
\text{mtype}(m, C_0) = \forall \overline{x : T}. P \multimap E \quad i = \text{effectsAllowed}(P) \quad \text{receiver packed}}{\Gamma; \Delta \vdash^i t_0.m(\overline{t}) : [t_0/\text{this}][t_0/\text{this}_{\text{fr}}]\overline{[t/\bar{x}]E}} \text{ P-CALL} \\
\\
\frac{\Gamma \vdash \overline{t : T} \quad \Gamma; \Delta \vdash [\text{super}/\text{this}_{\text{fr}}]\overline{[t/\bar{x}]P} \quad C \text{ extends } C' \\
\text{mtype}(m, C') = \forall \overline{x : T}. P \multimap E \quad i = \text{effectsAllowed}(P) \quad \text{receiver packed}}{\Gamma; \Delta \vdash_C^i \text{super}.m(\overline{t}) : [\text{super}/\text{this}_{\text{fr}}]\overline{[t/\bar{x}]E}} \text{ P-SUPER} \\
\\
\frac{\Gamma; \Delta \vdash t : \exists x : T_i. P \quad \Gamma; \Delta' \vdash_C [f_i/x']P' \otimes p \\
\text{localFields}(C) = \overline{f : T \text{ in } n} \quad n_i \leq n \quad p = \text{unpacked}(n, g, k, A), k \neq 0}{\Gamma; (\Delta, \Delta') \vdash_C^1 \text{assign } f_i := t : \exists x' : T_i. P' \otimes [f_i/x]P \otimes p \setminus f_i} \text{ P-ASSIGN}
\end{array}$$

Figure 18. Permission checking for expressions (part 2)

$$\begin{array}{c}
\frac{p = \text{access}(r, n, g, k, A)}{\text{purify}(p) = \text{pure}(r, n, g, A)} \quad \frac{\text{purify}(P_1) = P'_1 \quad \text{purify}(P_2) = P'_2 \quad \text{op} \in \{\otimes, \&, \oplus\}}{\text{purify}(P_1 \text{ op } P_2) = P'_1 \text{ op } P'_2} \\
\frac{\text{unit} \in \{\mathbf{1}, \top, \mathbf{0}\}}{\text{purify}(\text{unit}) = \text{unit}} \quad \frac{\text{purify}(P) = P'}{\text{purify}(\exists z : H. P) = \exists z : H. P'} \quad \frac{\text{purify}(P) = P'}{\text{purify}(\forall z : H. P) = \forall z : H. P'}
\end{array}$$

Figure 19. Permission purification

5.4 Field Assignments

Assignments to fields change the state of the receiver's current frame. We point out that assignments to a field do *not* change states of objects referenced by the field. Therefore reasoning about assignments mostly has to be concerned with preserving invariants of the receiver. The unpacked predicates introduced in section 5.2 help us with this task.

Our intuition is that assignment to a field requires unpacking the surrounding object to the point where all states that refer to the assigned field in their invariants are revealed. Notice that the object does not have to be unpacked completely in this scheme. For simplicity, each field is annotated with the subtree that can depend on it (figure 12). Thus we interpret subtrees as data groups [27].

The rule P-ASSIGN (figure 18) assigns a given object t to a field f_i and returns the old field value as an existential x' . This preserves information about that value. The rule verifies that the new object is of the correct type and that a suitable full or share permission is currently unpacked. By recording an effect on f_i we ensure that information about the old field value cannot "flow around" the assignment (which would be unsound).

5.5 Permission Reasoning with Splitting and Joining

Our permission checking rules rely on proving a predicate P given the current valid and linear resources, written $\Gamma; \Delta \vdash P$. We use standard rules for the decidable multiplicative-additive fragment of linear logic (MALL) with quantifiers that only range over fractions [28]. Following Boyland [7] we introduce a notion of substitution into the logic that allows substituting a set of linear resources with an equivalent one.

$$\frac{\Gamma; \Delta \vdash P' \quad P' \Rightarrow P}{\Gamma; \Delta \vdash P} \text{ SUBST}$$

The judgment $P \Rightarrow P'$ defines legal substitutions. We use substitutions for splitting and joining permissions (figure 20). The symbol \Leftrightarrow indicates that transformations are allowed in both directions. SYM and ASYM generalize the rules from section 2. Most other rules are used to split permissions for larger subtrees into smaller ones and vice versa. A detailed explanation of these rules can be found in the companion report [5].

Our splitting and joining rules maintain a consistent set of permissions for each object so that no permission can ever violate an assumption another permission makes. Fractions

$$\begin{array}{c}
\frac{A = A' = A'' \text{ or } (A = A' \text{ and } A'' = n) \text{ or } (A = A'' \text{ and } A' = n)}{\text{access}(r, n, g, k, A) \iff \text{access}(r, n, g/2, k/2, A') \otimes \text{access}(r, n, g/2, k/2, A'')} \text{ SYM} \\
\frac{A = A' = A'' \text{ or } (A = A' \text{ and } A'' = n) \text{ or } (A = A'' \text{ and } A' = n)}{\text{access}(r, n, g, k, A) \iff \text{access}(r, n, g/2, k, A') \otimes \text{pure}(r, n, g/2, A'')} \text{ ASYM} \\
\frac{n_1 \# n_2 \quad A_1 \prec n_1 \leq n \quad A_2 \prec n_2 \leq n}{p_i = \text{full}(r, n_i, \{g, \text{nodes}(n_i, n) \mapsto 1\}/2, A_i)} \text{ F-SPLIT-}\otimes \\
\frac{p_i = \text{full}(r, n_i, \{g, n \mapsto 1, \text{nodes}(n_i, n) \mapsto 1\}/2, A_i)}{p_1 \otimes p_2 \Rightarrow \text{full}(r, n, \{g, n \mapsto 1\}, A_1 \otimes A_2)} \text{ F-JOIN-}\otimes \\
\frac{A_1 \# A_2}{\text{full}(r, n, g, A_1 \oplus A_2) \iff \text{full}(r, n, g, A_1) \oplus \text{full}(r, n, g, A_2)} \text{ F-}\oplus \\
\frac{A \prec n' \leq n}{\text{full}(r, n, g, A) \Rightarrow \text{full}(r, n', \{g, \text{nodes}(n', n) \mapsto 1\}, A)} \text{ F-DOWN} \\
\frac{A \prec n' \leq n}{\text{full}(r, n', \{g, n \mapsto 1, \text{nodes}(n', n) \mapsto 1\}, A) \Rightarrow \text{full}(r, n, \{g, n \mapsto 1\}, A)} \text{ F-UP} \\
\frac{n' \leq n}{\text{pure}(r, n, \{g, \text{nodes}(n', n) \mapsto \bar{k}\}, A) \Rightarrow \text{pure}(r, n', g, A)} \text{ P-UP} \\
\frac{}{\text{access}(r, n, g, k, A) \Rightarrow \text{access}(r, n, g, k, n)} \text{ FORGET}
\end{array}$$

Figure 20. Splitting and joining of access permissions

of all permissions to an object sum up to (at most) 1 for every node in the object’s state space.

5.6 Example

To illustrate how verification proceeds, figure 21 shows the `fill` method from `BufferedInputStream` (figure 11) written in our core language. As can be seen we need an intermediate state `reads` and a marker field `reading` that indicate an ongoing call to the underlying stream. We also need an additional state refinement to specify an internal method replacing the `while` loop in the original implementation. (We assume that `thisfr` permissions can be used for calls to `private` methods.)

Maybe surprisingly, we have to reassign field values after `super.read()` returns. The reason is that when calling `super` we lose temporary state information for `this`. Assignment re-establishes this information and lets us pack properly before calling `doFill` recursively or terminating in the cases of a full buffer or a depleted underlying stream.

It turns out that these re-assignments are *not* just an inconvenience caused by our method but point to a real problem in the Java standard library implementation. We could implement a malicious underlying stream that calls back into the “surrounding” `BufferedInputStream` object. This call changes a field, which causes the buffer’s invariant on `count` to permanently break, *later on* resulting in an undocumented array bounds exception when trying to read behind the end of the buffer array.

Because `fill` operates on a share permission our verification approach forces taking into account possible field changes through reentrant calls with other share permissions. (This is precisely what our malicious stream does.) We could avoid field re-assessments by having `read` require a full permission, thereby documenting that reentrant (modifying) calls are not permitted for this method.

6. Related Work

In previous work we proposed more expressive typestate specifications [4] that can be verified with the approach presented in this paper. We also recently proposed full and pure permissions and applied our approach to specifying full Java iterators [3]. Verification of protocol compliance has been studied from many different angles including type systems, abstract interpretation, model checking, and verification of general program behavior. Aliasing is a challenge for all of these approaches.

The system that is closest to our work is Fugue [12], the first modular typestate verification system for object-oriented software. Methods are specified with a deterministic state transition of the receiver and pre-conditions on arguments. Fugue’s type system tracks objects as “not aliased” or “maybe aliased”. Leveraging research on “alias types” [33] (see below), objects typically remain “not aliased” as long as they are only referenced on the stack. Only “not aliased” objects can change state; once an object becomes “maybe

```

class BufferedInputStream extends FilterInputStream {
    states ready, reads refine open; ...
    states partial, complete refine filled;

    reads := reading; ready := reading = false; ...

    private boolean reading; ...

    public int read() : ∀k : Fract.... =
        unpack(open, k) in
            let r = reading in if(r == false, ... fill() ...)

    private bool fill() : ∀k : Fract.
        share(thisfr, open) in depleted ⊕ eof —
        share(thisfr, open) in available ⊕ eof =
        unpack(open, k, depleted ⊕ eof) in
            assign count = 0 in assign pos = 0 in
            assign reading = true in
            pack to reads in
            let b = super.read() in
            unpack(open, k, open) in
                let r = reading in assign reading = false in
                assign count = 0 in assign pos = 0 in
                if(r, if(b = -1, pack to eof in false,
                        pack to depleted in doFill(b)),
                   pack to eof in false)

    private bool doFill(int b) : ∀k : Fract.
        share(thisfr, open) in depleted ⊕ partial —
        share(thisfr, open) in partial ⊕ complete =
        unpack(open, k, depleted ⊕ partial) in
            let c = count in let buffer = buf in
            assign buffer[c] = b in assign count = c + 1 in
            let l = buffer.length in
            if(c + 1 >= l, pack to complete in true,
               assign reading = true in pack to reads in
               let b = super.read() in unpack(open, k) in
                   let r = reading in assign reading = false in
                   assign count = c + 1 in assign pos = 0 in
                   pack to partial in
                       if(r == false || b == -1, true, doFill(b))

```

Figure 21. Fragment of `BufferedInputStream` from figure 11 in core language

“aliased” its state is permanently fixed although fields can be assigned to if the object’s abstract typestate is preserved.

Our work is greatly inspired by Fugue’s abilities. Our approach supports more expressive method specifications based on linear logic [18]. Our verification approach is based on “access permissions” that permit state changes even in the presence of aliases. We extend several ideas from Fugue to work with access permissions including state invariants, packing, and frames. Fugue’s specifications are expressible with our system [4]. Fugue’s “not aliased” objects can be simulated with unique permissions for alive and “maybe aliased” objects correspond to share permissions with state guarantees. There is no equivalent for state dimensions, tem-

porary state assumptions, full, immutable, and pure permissions, or permissions for object parts in Fugue.

Verification of protocol compliance has also been described as “resource usage analysis” [23]. Protocol specifications have been based on very different concepts including typestates [34, 11, 25], type qualifiers [16], size properties [9], direct constraints on ordering [23, 35], and type refinements [30, 10]. None of the above systems can verify implementations of object-oriented protocols like our approach and only two [35, 10] target object-oriented languages. Effective type refinements [30] employ linear logic reasoning but cannot reason about protocol implementations and do not support aliasing abstractions. Hob [25] verifies data structure implementations for a procedural language with static module instantiation based on typestate-like constraints using shape analyses. In Hob, data can have states, but modules themselves cannot. In contrast, we can verify the implementation of stateful objects that are dynamically allocated and support aliasing with permissions instead of shape analysis. Finally, concurrent work on Java(X) proposes “activity annotations” that are comparable to full, share, and pure permissions for whole objects that can be split but not joined. Similar to effective type refinements, state changes can be tracked for a pre-defined set of types, but reasoning about the implementation of these types is not supported. To our knowledge, none of the above systems supports temporary state information.

Because programming with linear types [36] is very inconvenient, a variety of relaxing mechanisms were proposed. Uniqueness, sharing, and immutability (sometimes called read-only) [7] have recently been put to use in resource usage analysis [23, 9]. Alias types [33] allow multiple variables to refer to the same object but require a linear token for object accesses that can be borrowed [7] during function calls. Focusing can be used for temporary state changes of shared objects [13, 16, 2]. Adoption prevents sharing from leaking through entire object graphs (as in Fugue [12]) and allows temporary sharing until a linear adopter is deallocated [13]. All these techniques need to be aware of all references to an object in order to change its state.

Access permissions allow state changes even if objects are aliased from unknown places. Moreover, access permissions give fine-grained access to individual data groups [27]. States and fractions [6] let us capture alias types, borrowing, adoption, and focus with a single mechanism. Sharing of individual data groups has been proposed before [7], but it has not been exploited for reasoning about object behavior. In Boyland’s work [6], a fractional permission means immutability (instead of sharing) in order to ensure non-interference of permissions. We use permissions to keep state assumptions consistent but track, split, and join permissions in the same way as Boyland.

Global approaches are very flexible in handling aliasing. Approaches based on abstract interpretation (e.g. [1, 19, 14]) typically verify client conformance while the protocol implementation is assumed correct. Sound approaches rely on a global aliasing analysis [1, 14]. Likewise, most

model checkers operate globally (e.g. [21]) or use assume-guarantee reasoning between coarse-grained static components [17, 22]. The Magic tool checks individual C functions but has to inline user-provided state machine abstractions for library code in order to accommodate aliasing [8]. The above analyses typically run on the complete code base once a system is fully implemented and are very expensive. Our approach supports developers by checking the code at hand like a typechecker. Thus the benefits of our approach differ significantly from global analyses.

Recently, there has been progress in inferring typestate protocols in the presence of aliasing [31], which we believe could be fruitfully combined with our work to reduce initial annotation burden.

Finally, general approaches to specifying program behavior [26, 15, 2] can be used to reason about protocols. The JML [26] is very rich and complex in its specification features; it is more capable than our system to express object behavior (not just protocols), but also potentially more difficult to use due to its complexity. Verifying JML specifications is undecidable in the general case. Tools like ESC/Java [15] can partially check JML specifications but are unsound because they do not have a sound methodology for handling aliasing. Spec# is comparable in its complexity to the JML and imposes similar overhead. The Boogie methodology allows sound verification of Spec# specifications but requires programs to follow an ownership discipline [2].

Our system is much simpler than these approaches, focusing as it does on protocols, and it is designed to be decidable. Our treatment of aliasing makes our system sound, where ESC/Java is not. While the treatment of aliasing in our system does involve complexity, it gives the programmer more flexibility than Boogie's while remaining modular and sound. Because it is designed for protocol verification in particular, our system will generally impose smaller specification overhead than the JML or Spec#.

7. Conclusions

This paper proposes a sound modular protocol checking approach, based on typestates, that allows a great deal of flexibility in aliasing. A novel abstraction, access permissions, combines typestate and object aliasing information. Developers express their protocol design intent using access permissions. Our checking approach then tracks permissions through method implementations. For each object reference the checker keeps track of the degree of possible aliasing and is appropriately conservative in reasoning about that reference. A way of breaking an invariant in a frequently used Java standard library class was exposed in this way. The checking approach handles inheritance in a novel way, giving subclasses more flexibility in method overriding. Case studies on Java iterators and streams provide evidence that access permissions can model realistic protocols, and protocol checking based on access permissions can be used to reason precisely about protocols arising in practice.

In future work we hope to further refine and evaluate our approach. We plan to develop a deterministic algorithm

for reasoning about permissions. We hope to leverage our experiences in using our approach to increase its practicality. Based on the case studies presented in this paper we made the following observations:

- In this paper we chose to make the linear logic formalism underlying our approach explicit in example protocol specifications. However, our case studies suggest that practical protocols follow certain patterns. For example, method specifications often consist of simple conjunctions that can be expressed by annotating each method argument separately. With syntactic sugar for such patterns we believe that programmers will only rarely have to use linear logic operators explicitly.
- Specification effort lies primarily with protocol *implementation* developers, which better amortizes over time. Conversely, iterator, stream, and other libraries' clients have (we believe) minimal work to do unless they store objects in fields. (Fugue's experience suggests that loop invariants for typestate checking can often be inferred [12].)
- Only a fraction of our system's capabilities are needed for any given example (although they all are necessary in different situations). Developers do have to understand the general idea of access permissions.

We believe that these observations indicate that the approach can be practical, especially with the help of syntax that captures common cases concisely. A systematic evaluation of this claim is an important part of planned future work.

Acknowledgments

We thank John Boyland, Frank Pfennig, the Plaid group, Sebastian Boßung, and Jason Reed for fruitful discussions on this topic. We also thank the anonymous reviewers for their helpful feedback. This work was supported in part by NASA cooperative agreement NNA05CS30A, NSF grant CCF-0546550, the Army Research Office grant number DAAD19-02-1-0389 entitled “Perpetually Available and Secure Information Systems”, and the U.S. Department of Defense.

References

- [1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of the Eighth SPIN Workshop*, pages 101–122, May 2001.
- [2] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.
- [3] K. Bierhoff. Iterator specification with typestates. In *5th Int. Workshop on Specification and Verification of Component-Based Systems*, pages 79–82. ACM Press, Nov. 2006.
- [4] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *Joint European Software Engineering Conference and ACM Symposium on the Foundations of*

- Software Engineering*, pages 217–226. ACM Press, Sept. 2005.
- [5] K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. Technical Report CMU-ISRI-07-105, Carnegie Mellon University, Mar. 2007. <http://reports-archive.adm.cs.cmu.edu/anon/isri2007/CMU-ISRI-07-105.pdf>.
- [6] J. Boyland. Checking interference with fractional permissions. In *Int. Symposium on Static Analysis*, pages 55–72. Springer, 2003.
- [7] J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *ACM Symposium on Principles of Programming Languages*, pages 283–295, Jan. 2005.
- [8] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Int. Conference on Software Engineering*, pages 385–395, May 2003.
- [9] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popescu, and H. H. Nguyen. Verifying safety policies with size properties and alias controls. In *Int. Conference on Software Engineering*, pages 186–195, May 2005.
- [10] M. Degen, P. Thiemann, and S. Wehr. Tracking linear and affine resources with Java(X). In *European Conference on Object-Oriented Programming*. Springer, Aug. 2007.
- [11] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, 2001.
- [12] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.
- [13] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.
- [14] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ACM Int. Symposium on Software Testing and Analysis*, pages 133–144, July 2006.
- [15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, May 2002.
- [16] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2002.
- [17] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Int. Conference on Software Engineering*, pages 211–220, May 2004.
- [18] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [19] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, 2002.
- [20] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8:231–274, 1987.
- [21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 58–70, 2002.
- [22] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *ACM Int. Symposium on Software Testing and Analysis*, pages 39–49. ACM Press, July 2007.
- [23] A. Igarashi and N. Kobayashi. Resource usage analysis. In *ACM Symposium on Principles of Programming Languages*, pages 331–342, Jan. 2002.
- [24] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 132–146, 1999.
- [25] V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), Dec. 2006.
- [26] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
- [27] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 144–153, Oct. 1998.
- [28] P. Lincoln and A. Scedrov. First-order linear logic without modalities is NEXPTIME-hard. *Theoretical Computer Science*, 135:139–154, 1994.
- [29] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
- [30] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ACM Int. Conference on Functional Programming*, pages 213–225, 2003.
- [31] M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 77–96, 2005.
- [32] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *ACM Conference on Programming Language Design and Implementation*, pages 83–94, 2002.
- [33] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381. Springer, 2000.
- [34] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.
- [35] G. Tan, X. Ou, and D. Walker. Enforcing resource usage protocols via scoped methods. In *Int. Workshop on Foundations of Object-Oriented Languages*, 2003.
- [36] P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.