CORRECT BY CONSTRUCTION LANGUAGE IMPLEMENTATIONS

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus Prof.dr.ir. T.H.J.J. van der Hagen
chair of the Board of Doctorates
to be defended publicly on
Thursday 14 October 2021 at 15:00 o'clock

by

Arie Jonathan ROUVOET
Master of Science in Computer Science,
Delft University of Technology, the Netherlands
born in Nunspeet, the Netherlands

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	Chairperson
Prof.dr. E. Visser	Delft University of Technology, promotor
Dr.ir. R.J. Krebbers	Radboud University and Delft University of Technology, copromotor

Independent members:

Prof.dr. P. Wadler	University of Edinburgh, United Kingdom
Prof.dr. J. Gibbons	University of Oxford, United Kingdom
Prof.dr. G.K. Keller	Utrecht University
Prof.dr. A. van Deursen	Delft University of Technology
Prof.dr. J.A. Perez Parra	University of Groningen
Prof.dr.ir. D.H.J. Epema	Delft University of Technology (reserve member)

ISBN: 978-94-6384-256-3

The work in this thesis has been carried out at the Delft University of Technology under the auspices of the research school IPA (Institute for Programming research and Algorithmics). This research was funded by the NWO VICI Language Designer's Workbench project (639.023.206).

Voor opa Arie en oma Truus Rouvoet

Bit by bit, putting it together.
Piece by piece, only way to make a work of art.
Every moment makes a contribution
Every little detail plays a part
Having just a vision's no solution
Everything depends on execution
Putting it together, that's what counts!
Ounce by ounce, putting in together.
Small amounts, adding up to make a work of art
First of all you need a good foundation
Otherwise it's risky from the start
Takes a little cocktail conversation
But without the proper preparation
Having just a vision's no solution
Everything depends on execution
The art of making art
Is putting it together, bit by bit.
Shpiel by shpiel, doubt by doubt.
And that... is the state of the art!

— Barbara Streisand,
Putting it Together, 1985

Acknowledgments

The chance to work for four years at a university on an interesting research topic is really marvelous and I consider it a great privilege. I am thankful to Sandro Stucki, Erik Meijer and Eelco Visser, for supervising my master thesis and their part in getting me started on this trajectory.

From quite early on I chose my own directions within the topic to which I was appointed. Thank you, Eelco, for accommodating that. The freedom has taught me a lot. At the same time you were engaged with my projects and you have corrected my course at the right times. We have debated many things, and much of my progress on the topic and in my thinking can be attributed to that. With both your work in the fore- and background, you have created the space for me to learn. With that, your impact on my development has been profound.

Likewise, Robbert. You were paramount in making this thesis possible. Throughout you were available for any question, open to any idea, and interested in any peculiarity. Your technical knowledge and good eye for detail have been an inspiration and an important resource that the contributions in this thesis are building upon.

Thanks, Casper and Hendrik, for collaborating on various projects and for being my senior colleagues from whose experience I was allowed to draw and learn! Thanks Daco, Peter, Jules, Cas, and Paolo for being great office mates, good climbers, and willing chess victims. Thanks you, Andrew Tolmach, for our fruitful collaboration and for feedback on various drafts and ideas! Many thanks to all the members of the doctoral committee, and especially to Phil Wadler, Jeremy Gibbons, and Jorge Perez for providing much valuable feedback on the submitted draft of this thesis. To Roniet, for all the innumerable things that you have looked up, filed, negotiated, kept afloat, and organized. To all my colleagues: thanks for being terrific coffee company.

Before any of this happened, however, my lovely, wise and kind significant other was already present and helping me figure out whether to undertake this journey at all. Irene, thank you for the patience, the good advise, the consolation. Thank you for good times. If not with you, then not at all. Let's stay together and live in the faith that God will provide for us, just as certain as that He has provided to this day.

Behind every great wife, there are great parents in law. Thank you, Andre and Hannelie, for maintaining a safe haven where we can always drop by to catch up with life.

Then, to my own fantastic mum and dad, André and Liesbeth. Your contributions start before my first memories, so I will not attempt to name them. Thanks especially for all the support throughout these last four-ish years. When I really worry, I fall back on knowing that we can always rely on you for anything.

Finally, to my friends and family in the Netherlands, Canada, South Africa, Sweden and Denmark: you are good people. Thanks for hanging about with me.

"Look at the birds.
They don't plant
or harvest
or store food in barns,
for your heavenly Father feeds them.
And aren't you far more valuable to him than they are?
Can all your worries add a single moment to your life?"

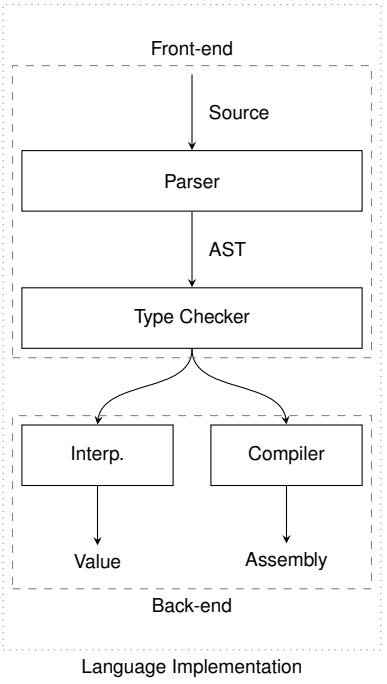
— Matthew 6:25-27

Contents

1 Introduction

> "Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning."
>
> — Hoare (1969)

Implementations of *typed* programming languages consist of a front-end and a back-end. The purpose of the front-end is to determine whether the textual input is indeed a meaningful program. The parser first determines if the program is syntactically valid, producing an abstract syntax tree (AST) that represents the program. The function of the type checker is to check if the AST is also meaningful. Concrete tasks of type checkers are resolving identifiers, checking that declared and actual types match, etc. Some of the accrued information will be stored as annotations on the AST so that the back-end has access to it.



When the type checker decides to accept the program, the program is judged 'meaningful'. The back-end assigns this meaning—i.e., the dynamic semantics—by making the program executable. An interpreter is a language back-end that directly evaluates the program—e.g., by reducing it step by step according to the reduction rules of the language. A compiler back-end indirectly makes the program executable by transforming it into a program in a (usually) low-level language such as Java Virtual Machine bytecode. The low-level program can be executed by the actual or virtual machine.

The decision of the type checker reflects a contract between the programmer and the programming language implementation. It is up to the programmer to deliver a program that is well typed, but it is up to the language implementation to execute well-typed

programs without crashing unexpectedly.[1] This thesis is about implementing type checkers, interpreters, and compilers in such a way that each, *by construction*, fulfills their role in maintaining this contract.

[1] There is some wiggle room here in the interpretation of what amounts to an 'unexpected' crash. The contract of a typed programming language should specify this.

The type contract induces *safety* criteria for the type-checker and the different back-ends of a language that ensure that type errors will never occur when a program is executed. These criteria are formulated relative to a specification of program *typing*. The specification of program typing is also called a *static semantics* of a programming language. The criteria are as follows:

- A type checker shall only accept programs that have a typing according to the specification of the static semantics. A type checker that has this property is said to be *sound* with respect to the static semantics.[2]
- An interpreter must reduce well-typed expressions to values of *the same type*. An interpreter that has this property is said to be *type safe*.
- A compiler transforms well-typed source expressions to well-typed target expressions of a corresponding type. In other words, compilers preserve well-typing, extending the contract on the input to the output. Compilers that satisfy this property are said to be *type correct*.

[2] In addition to type checker soundness, one might desire a type checker to also be *complete*, so that it only rejects programs that do not have a typing. Completeness is not a safety criterion, because programs that are unnecessarily rejected will never be executed.

These safety criteria are essential to ensure that the typing of a source program is a meaningful contract. A type-checker that violates soundness incorrectly assigns types to programs that are in fact ill typed, according to the specification of the static semantics. An interpreter that violates type-safety can reduce an expression of some type to a value of a different type or get stuck, thus nullifying the usefulness of checking that the source program is well typed. Similarly, a compiler that violates type correctness can transform a well-typed source program into an ill-typed target program, again negating the point of checking types on the source language.

The importance of the above criteria raises the question how a language developer can establish that these criteria are satisfied by the implementation. Testing can famously only prove the presence of bugs.[3] To prove that the language implementation satisfies

[3] Dijkstra 1972. "Notes on structured programming"

the criteria on all inputs, one must turn to *formal verification*—i.e., by specifying it unambiguously in a *formal language* and giving a derivation that proves that it holds.

A *formal* language or system is one that operates by well-defined rules. Programming languages are good examples of formal systems, because their static and dynamic semantics can be specified as sets of rules:[4] static semantics is specified using typing rules, and dynamic semantics using evaluation rules. Such rule-based specifications lend themselves well to *formal reasoning*, in contrast, for example, to specifications in natural language. To prove something *in* such a formal system is to give a *derivation* of rule applications. For example, we can prove that a particular program is well typed by delivering a type derivation.

[4] Hoare 1969. "An Axiomatic Basis for Computer Programming"

To formally verify a property *about* a formal system $\mathcal{S}$ we repeat the trick one level higher: we write down the rules of $\mathcal{S}$ in a formal *meta language* $\mathcal{M}$. We then use the rules of the meta language $\mathcal{M}$ to prove some property about, for example, all derivations in $\mathcal{S}$. To distinguish the formal system being studied from the formal system in which we study it, we sometimes speak of an *object language* $\mathcal{S}$.

How then should a language developer *prove* that their language front-ends and back-ends satisfy the safety criteria? Conventional proofs by language researchers consist of separate specifications, implementations, and a manual proof. This thesis is driven by a vision of a more integrated approach: language specifications and/or implementations ought to be written in meta-languages that *ensure* the relevant safety properties, avoiding somehow the need for separate proofs for those properties.

There are different approaches to accomplishing this goal. We investigate two such approaches in this thesis. For language front-ends, we investigate an approach where the language developer specifies the static semantics in a meta-language which automatically delivers the implementation of a sound type checker. For language back-ends, we investigate an approach where the language developer implements an interpreter or compiler using a meta-language that is *intrinsically typed*—i.e., if the implementation type-checks in the meta-language, then it satisfies the type-correctness criterion.

In both cases, we develop meta-languages that accommodate *high-level* specifications. That is, specifications that are precise, but also aim to be "sufficiently simple to be understood both by the [language developer] and by a reasonably sophisticated user of the

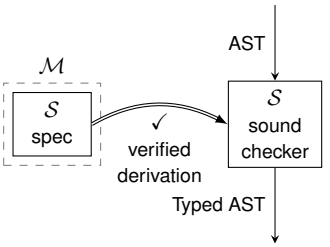
[object] language"[5]. Hoare (1969) states that we can only maximize the advantage from a formal language specification if it bridges the communication gap between language developers and users. We pursue this idea, but also go beyond it. Formal language specifications should also be *instrumental* in developing programming language implementations that are correct by construction.

In sections 1.1 and 1.2, we explain the problems of existing approaches, as well as the main ingredients and challenges of the proposed approaches. We end the introduction with an overview of the contributions of this thesis.

[5] Hoare 1969. "An Axiomatic Basis for Computer Programming"

1.1 *Safe Language Front-Ends*

How should a language developer ensure that their language's type checker is sound with respect to the specification of the language's $\mathcal{S}$ static semantics? The perspective that we take in part II of this thesis, is that it is possible to systematically derive type checkers from certain specifications of static semantics and to prove once-and-for-all that the derivation always delivers *sound* type checkers.[6]



[6] This is analogous to how we treat parsing: a language developer specifies a grammar in a suitable meta-language, and the parser for that grammar is systematically derived.

What then is a suitable meta-language $\mathcal{M}$ in which we can write those specifications? A specification of static semantics should be *high-level*: it describes a contract between the language implementation and the programmer, so it should explain the typing of programs using concepts that programmers can understand. This in particular precludes all of the *low-level* implementation details of a type checker, such as the maintenance of symbol tables.

Although it is well understood how to specify the static semantics of many core features of programming languages, the surface syntax of many actual programming languages are still hard to specify formally and at a high-level.[7] This is a first challenge for the design of the meta-language $\mathcal{M}$: *what are high-level concepts for the specification of static semantics for surface-level language features*?

[7] Which may well contribute to the remarkable fact that almost none of the programming languages have formal specifications of the static semantics of their surface-level features, with the notable exception of ML (Milner et al. 1997).

At the same time however, we want to be able to automatically derive a type checker from the specification. This causes tension in the design of the specification language. We need to identify concepts that are both high-level, and that enable the recovery of all the necessary low-level details of a sound implementation. This is a second challenge for the design of the meta-language $\mathcal{M}$: *is it feasible to derive implementations from high-level specifications and prove the derivation sound once-and-for-all*?

One particular programming language feature that is worthy of attention with respect to that challenge, is *static binding*. Despite the great variety of static binding in programming languages, only λ- or let-binding can be concisely and precisely specified using the context-expression-type ($\Gamma \vdash e : t$) presentation of static semantics that are used almost exclusively in research papers. In contrast, even a simple program in languages such as ML, Java, or Scala can use identifiers in different name spaces, type-based name resolution, (first-class) modules and imports, or other language specific scoping and resolution mechanisms. Many of those involve scoped global static binding, which cannot be specified using mere lexical typing contexts.[8]

[8] The definition of Standard ML (Milner et al. 1997) uses a natural semantics to specify static semantics with both input and output environments. As a consequence, the specification is more operational in character.

At the same time, the implementation of a type checker for a language such as ML, Java, or Scala requires some *scheduling* of type checking tasks: more ways to refer to other parts of the program cause more possible dependencies between type checking tasks. Language developers manually stratify the type checking algorithm into various passes (e.g., Haskell, ML), or employ more dynamic scheduling of type checking tasks (e.g., Rust). In either case, their strategy must ensure that a name is never resolved before all relevant definitions have been recorded, all relevant imports are resolved, and all relevant types have been checked, in order for the type checker to be sound.

In part II of this thesis we propose the meta-language Statix. In Statix, a specification of static semantics is defined using a typing relation. The main new feature of Statix is primitive premises for the specification of static binding. In particular, it has premises for asserting unique nodes and edges in a global *scope graph*[9], which can be understood as a structured symbol table. The idea to use graphs to model binding comes from the observation that static binding in program abstract syntax trees can be understood as additional edges relating references to declarations, leading to graph-like descriptions of programs. The additional structural information can be exploited to give concise specifications of the static binding in for example module systems and object-oriented languages.[10,11]

[9] Neron et al. 2015. "A Theory of Name Resolution"

[10] Van Antwerpen et al. 2016. "A constraint language for static semantic analysis based on scope graphs"

[11] Hedin 2000. "Reference Attributed Grammars"

Typing rules in Statix have a *declarative semantics* as inference rules of a separation logic[12,13] that explains how the global symbol table (i.e., the scope graph) is made up of individual entries coming from declarations in the program AST. It is the separation logic

[12] Reynolds 2002. "Separation logic: A logic for shared mutable data structures"

[13] O'Hearn et al. 2001. "Local Reasoning about Programs that Alter Data Structures"

flavor of the rules that enables the concise rules for specifying global structure.

To obtain a type checker from a Statix specification, we equip Statix with a non-backtracking *operational semantics*, inspired by the operational semantics of Constraint Handling Rules[14]. The problem that we face is to give a general strategy that ensures that scope graph queries modeling name resolution are only resolved after all relevant scoping structure has been recorded. Because scoping structure may depend on name resolution, this requires a non-trivial strategy. We give a strategy and prove that the resulting type checkers are sound with respect to the declarative semantics of the Statix specification.

[14] Frühwirth 1998. "Theory and Practice of Constraint Handling Rules"

Using the Statix meta-language we are able to specify the surface-level static binding of, for example, class hierarchies in Java, and the imports from objects in Scala. The operational semantics can evaluate these specifications on a test suite of Java and Scala programs, and delivers verdicts that agree with the type checkers of those languages.

1.2 *Type-Correct Language Back-Ends*

How should a language developer ensure that their interpreter is type-safe and their compiler is type-correct? An appealing approach is to use *intrinsically typed functional definitions*, because they are executable and type-safe by construction. This section positions this approach relative to conventional (syntactic) safety proofs and explains the contributions of part I of this thesis.

In 1978, Milner coined the slogan "well-typed programs cannot *go wrong*" characterizing (weak) *type soundness*.[15] The subject of the theorem in his paper is a denotational semantics of a typed functional language. This semantics is characterized by a meaning function that maps into a domain that includes a denotation "wrong" which is used to give meaning to ill-typed programs. Milner proves weak soundness using a stronger soundness theorem, which coincides with what we already called type safety.

[15] Milner 1978. "A theory of type polymorphism in programming"

Defining a denotational semantics like Milner's can be difficult because it requires that one identifies an appropriate semantic domain. In 1994, Wright and Felleisen published their article "A Syntactic Approach to Type Soundness" in which they describe a simpler proof method for type safety that has become *the* method of proving

type soundness that we teach.[16,17,18] Their approach is based on step-wise reduction of expressions $e \longrightarrow e'$ (e steps to e'). Instead of having a reduction into "wrong", ill-typed expressions simply do not reduce: the reduction relation $\longrightarrow$ gets stuck. To prove type safety, one proves a lemma called *subject reduction* or *(type) preservation*, which is based entirely on the typing of expressions:

$$\text{If } \vdash e : \tau \text{ and } e \longrightarrow e' \text{ then } \vdash e' : \tau.$$

That is, reducing a well-typed expression e of type τ to an expression e' preserves the type. This lemma is combined with a lemma called *progress*:

$$\text{If } \vdash e : \tau \text{ then either } e \text{ is a value, or } e \longrightarrow e'.$$

That is, every well-typed expression that is not a value is not stuck and can reduce. Together these lemmas ensure type soundness. By progress, any well-typed expression e that is not already a value can be reduced to an expression e'. If e' is a value, then we are done. If not, then we argue by preservation that e' is again well-typed, and we repeat the process. Hence, well-typed expressions cannot go wrong.

The syntactic approach to proving type safety works uniformly for a surprising variety of programming languages (Pierce 2002). For example, Harper proves polymorphic reference type safe by a simple syntactic argument,[19] whereas the denotational semantics of polymorphic references has traditionally been difficult.

Additionally, a syntactic type safety proof can be *mechanized* straightforwardly. That is, syntactic type safety can be stated and proven in a proof assistant in the same way that it is stated on paper, except that one cannot leave trivial cases as an exercise to the reader. This is the case, because both the step relation and the syntactic typing can be embedded in a proof assistant like Coq as an inductively defined relation and the reasoning about such relations is well understood and is amenable to automation. These attractive properties have made the syntactic approach to type safety the default.[20]

Although syntactic proofs of type safety can be conducted well in a proof assistant for a large number of languages, there are some drawbacks to the approach. In particular, there is a large gap

[16] Wright et al. 1994. "A Syntactic Approach to Type Soundness"

[17] Harper 2016. "Practical foundations for programming languages"

[18] Pierce 2002. "Types and programming languages"

[19] Harper 1994. "A Simplified Account of Polymorphic References"

[20] Semantic type safety proofs have recently regained popularity. One reason for this is the advance of techniques enabling abstraction over difficult technical details of a domain, such as step-indexing of propositions (Appel et al. 2007; Dreyer et al. 2011; Jung et al. 2018b). Another reason is that syntactic safety proofs cannot be used to reason about programs that are (temporarily) not syntactically well-typed (Jung et al. 2018a).

between a small-step operational semantics and an executable interpreter which does not operate by substitution. If we are primarily interested in language implementations, then it is more sensible to start with an executable specification of the dynamic semantics.

A *definitional interpreter*[21,22,23] defines the operational semantics of a language using a high-level evaluation function written in a total functional language. A great advantage of a functional specification is that it is readily executable, so that the language can easily be tested by a language developer. A disadvantage of a functional definition, is that it must explicitly address the fact that ill-typed programs cannot be evaluated. For example, an interpreter must eventually raise an exception when it evaluates $\mathsf{true} + 42$. Similarly, the implementation must deal with the possibility that it itself makes type errors. For example, an interpreter that evaluates $(18 + 3) + 21$ by first recursively evaluating the addition $18 + 3$ must account for the possibility that this results in something other than a number!

In a proof assistant that demands functions to be total, such possibility of failure cascades through the definition: an interpreter must also handle errors thrown by recursive invocations. This imposes overhead on the definition. This overhead in the definition of the back-end is more than a mere nuisance. It obfuscates the behavior of the back-end on well-typed terms, reducing the quality of the specification.[24]

One can argue—and indeed we will—that the overhead of errors due to ill-typed expressions in the back-end is unnecessary because the back-end of a language implementation is only ever supposed to encounter well-typed expressions. A proof of type-safety is evidence that the interpreter will never raise an error (and will consequently never have to handle an error) when executed on well-typed terms. In other words, the definition of the semantics has overhead because we are working in a meta-language that demands functions to be total. Ironically, we then proceed to prove that the function is actually total (modulo divergent evaluation) on the subset of its domain that we are actually interested in when we prove type safety!

To remedy the irony, we can instead define an *intrinsically typed interpreter* that operates only on well-typed terms and that internalizes the type-safety theorem. This involves two challenges: (1) to find a suitable representation of well-typed expressions, and (2) to define an interpreter for those expressions that encapsulates the

[21] Reynolds 1998a. "Definitional Interpreters for Higher-Order Programming Languages"

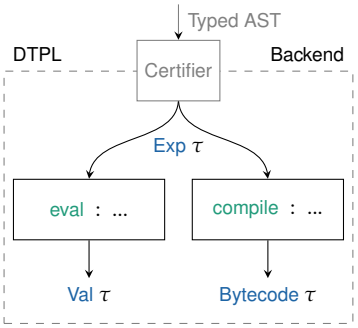
[22] Danielsson 2012. "Operational semantics using the partiality monad"

[23] Amin et al. 2017. "Type soundness proofs with definitional interpreters"

[24] It may even prompt us to specify the partial behavior on well-typed terms separately as a relation and prove an *adequacy* theorem of the interpreter, thus separating the specification from the implementation.

type-safety proof. The main challenge in accomplishing these is to ensure that the cure for partiality is not worse than the ailment: if the overhead of the proof exceeds the overhead of partiality, then we have accomplished nothing.

Encouragingly, previous work on intrinsically typed (definitional) interpreters[25] and also on intrinsically typed expression compilers[26] has shown that this is possible. This is accomplished by using a dependently typed functional programming language (DTPL) such as Agda[27] as a meta-language. Intrinsically typed terms are represented as an inductive family[28] (e.g., Exp τ) whose constructors effectively merge grammar with typing rules. An interpreter is represented by a function whose signature (e.g., eval : Exp τ → Val τ) expresses the safety theorem. The structure of syntactic proofs of type correctness is so similar to the structure of the interpreter that we superimpose them and get rid of half the work. This is what we call the *appeal* of intrinsically typed programming. Not only do we gain the ability to make use of type invariants in the implementation, we also eliminate most of the proof work.



[25] Augustsson et al. 1999. "An exercise in dependent types: A well-typed interpreter"

[26] McKinna et al. 2006. "A type-correct, stack-safe, provably correct, expression compiler"

[27] Norell 2008. "Dependently Typed Programming in Agda"

[28] Dybjer 1994. "Inductive Families"

Despite the appeal of intrinsically typed programming, it has not yet been adopted as broadly as Wright & Felleisen's syntactical approach. A major question is whether intrinsically typed programming scales beyond interpreters and compilers for simply typed expression languages. Programming languages with more elaborate type systems often have type invariants that are more subtle than strict preservation of types. For example, type-safe references cannot be typed in isolation, but must be typed relative to a runtime store. Consequently, the traditional syntactic type-safety proof for references requires additional arguments. It is unclear whether such type invariants and arguments can be integrated into an intrinsically typed interpreter with the same advantages as displayed in the simply typed cases.

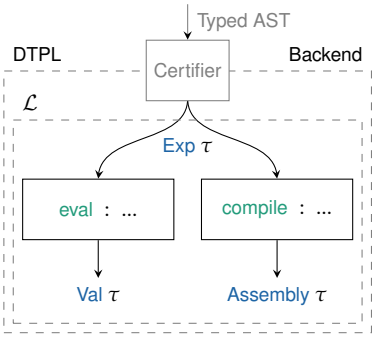
Part I of this thesis investigates the application of intrinsically typed programming to the following programs:

- An interpreter for a monomorphically typed functional language with ML-style references (chapter 3),
- An interpreter for a linear functional language with concurrency, and (session-) typed, cross-thread communication (chapter 4),
- A compiler targeting a bytecode language with only primitive

control flow using jumps to labeled instructions (chapter 5).

Indeed, we find that the key ideas that were sufficient for simply typed languages do not deliver the same advantages for these programs. The straightforward integration of the extrinsic type safety proofs results in programs that are littered with additional proof work that exceed the overhead of implementing their untyped, partial counterparts.

To address this problem, we investigate and deliver functional abstractions that not only encapsulate computational work but also *proof work*. We focus on abstractions that are in the spirit of intrinsically typed programming: the contribution of abstract operations to the proof that they participate in, is expressed in their type. To hide the details of the proofs alongside the details of the implementation, we work in various logics $\mathcal{L}$ embedded shallowly in the dependently typed meta-language.



In each chapter of this thesis we detail the technical contributions. The main contributions of part I can be summarized as follows:

- In order to intrinsically type an interpreter for a functional language with references, we construct a monotone state monad that avoids manual reasoning about a hidden monotone resource (e.g., a store). This yields typed programs with minimal overhead relative to the untyped, partial definitional interpreter that does not verify type-safety.

- In order to intrinsically type the syntax and operations of sub-structural languages, we contribute the idea of using an embedding of a novel proof-relevant separation logic, extending conventional (abstract) separation logic[29,30]. The separation logic successfully hides the tedious maintenance of both linear binding, and the invariants that are prevalent in the type-safety proofs of languages with typed sub-structural resources, such as session-typed channel references.

- In order to abstract over data and operations with sub-structural invariants we present a new algebraic structure of proof-relevant separation algebras (PRSAs), extending conventional separation algebras[31]. Every PRSA gives rise to a proof-relevant separation logic. By defining data structures and functional abstraction in the logic we gain reuse. Many familiar dependently typed programming devices readily have "resourceful" counterparts in

[29] O'Hearn et al. 2001. "Local Reasoning about Programs that Alter Data Structures"

[30] Calcagno et al. 2007. "Local action and abstract separation logic"

[31] Dockins et al. 2009. "A fresh look at separation algebras and share accounting"

the logic, which hide the accounting of some notion of resource usage.

- In order to give an intrinsically typed interpretation of linear references, we present a PRSA that balances demand for state (i.e., references) with the supply of state (e.g., cells in the store). The fact that the implementation of state maintains that richer sub-structural resource can be hidden from the clients of the state interface. We deliver an intrinsically typed interpreter for a linear functional language with concurrency, and (session-) typed, cross-thread communication
- In order to define a compositional, intrinsically typed compiler, we present a new nameless representation of labels in bytecode. Unlike named labels, nameless labels can be hidden without running the risk of defining it a second time, which would lead to ambiguous references. Compositionality is surprising, because the untyped counterpart using named labels threads state for generating unique label names.

With these contributions and their application to the show cases listed above, we deliver constructive evidence that intrinsically typed programming can scale well beyond simply typed languages. We manage not only to prove interesting invariants, but also to specify those invariants in such a way that they align well with computational abstractions of effects. We deliver interpreters and compilers that are total and have almost no proof overhead.

1.3 *Origin of the Chapters*

The contributions of both parts have previously been published in the refereed papers listed below. Although some of the content of the papers has been adapted for a consistent presentation in this thesis, these publications form the backbone of this thesis.

- C. B. Poulsen, A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser (2018). "Intrinsically-typed definitional interpreters for imperative languages". In: *Proceedings of the ACM on Programming Languages* 2.POPL, 16:1–16:34. DOI: 10.1145/3158104. Section 2.1 and chapter 3 are based on this paper, which was a collaborative effort. In addition to the work on interpreting type-safe reference that we present in chapter 3, this paper also describes an approach

for intrinsically typed global static binding using a library based on scope graphs. I do not present that in this thesis, because this approach can largely be attributed to my co-authors (see also Poulsen et al.[32], and section 7.1 for more discussion of the topic).

[32] Poulsen et al. 2016. "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics"

- A. Rouvoet, C. Bach Poulsen, R. Krebbers, and E. Visser (2020b). "Intrinsically-typed definitional interpreters for linear, session-typed languages". In: *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 284–298. DOI: 10.1145/3372885.3373818. Chapter 4 is based on this paper.
- A. Rouvoet, R. Krebbers, and E. Visser (2021). "Intrinsically-typed compilation with nameless labels". In: *Proceedings of the ACM on Programming Languages* 5.POPL, pp. 1–28. DOI: 10.1145/3434303. Chapter 5 is based on this paper.
- A. Rouvoet, H. Van Antwerpen, C. B. Poulsen, R. Krebbers, and E. Visser (2020a). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications". In: *Proceedings of the ACM on Programming Languages* 4.Object-Oriented Programming Systems, Languages & Applications (OOPSLA), 180:1–180:28. DOI: 10.1145/3428248. Chapter 6 is based on this paper, which was a collaborative effort.

During my PhD I also collaborated on the following refereed paper, which first introduces the specification language Statix that is also the subject of Rouvoet et al. 2020a (chapter 6).

- H. Van Antwerpen, C. B. Poulsen, A. Rouvoet, and E. Visser (2018). "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), 114:1–114:30. DOI: 10.1145/3276484

1.4 Mechanization of the Results

The contributions of part I have been implemented fully in the Agda proof-assistant and its source code is publicly available (see below). Although the code in this thesis is also typeset using the Agda type checker, it has undergone minor modifications and much has been omitted to serve the purpose of explaining the contributions of the thesis.[33] Hence, anyone who desires to replicate the work in Agda

[33] The purpose of the source code is not entirely the same as the purpose of this thesis, because it also aims to be a (re)usable library. This means that the source code is oftentimes more general and more modular than the presented code.

should make use of the available source code, which contains all the details in full, and more.

The Statix specifications language that is the subject of part II is implemented in Haskell and also publicly available, together with the case studies. The proofs about Statix have not been mechanized.

The repository that aggregates the source repositories for the various projects is available on github:

github.com/ajrouvoet/thesis-artifact

A snapshot version of this source package is archived on Zenodo.

PART I
INTRINSICALLY TYPED LANGUAGE IMPLEMENTATIONS

2 Verification of Language Implementations Using Dependent Types

> "There is a tendency to see programming as a fixed notion, essentially untyped. [...] This conception fails to engage with the full potential of types to make a positive contribution to program construction."
>
> — McBride (2004)

In chapter 1 we proposed the use of intrinsically typed programs to produce formally verified language back-ends. In this section, we present a brief introduction to dependently typed programming using Agda[1], a functional programming language in the style of Haskell with state-of-the-art support for programming with dependent types. We start with the basics of dependently typed languages using vectors as an example. We do not attempt to give a complete introduction to the subject[2,3,4], but highlight the aspects that are key to our approach. Readers that are familiar with dependent types in Agda can skip over this exposition and continue reading in section 2.2.

[1] The Agda name and logo are inspired by a Swedish song about a hen and a rooster. A rooster appears as the logo of another well known proof assistant.

[2] Norell 2008. "Dependently Typed Programming in Agda"

[3] Stump 2016. "Verified functional programming in Agda"

[4] Wadler et al. 2020. "Programming Language Foundations in Agda"

2.1 *Dependent Types*

In statically typed programming languages, types describe knowledge about the shape of the data in a program. We say that a type system of a language is *strong* when (a) it is feasible to express precise knowledge in the types, and (b) when the types are *reliable*. That is, the knowledge reflected in a type cannot turn out to be false at runtime.[5] A dependently typed language like Agda takes both aspects of strong type systems very seriously.

[5] The most famous unreliable type is the type of references in languages with `null` references. The type promises a reference to an object on the heap, but can turn out to be `null` at runtime.

FIRST, TYPES MUST BE RELIABLE. If not, then this is considered a bug in the implementation of Agda. This has far reaching consequences. For example, a function with the following type simply does not exist in Agda:

```
head : {A : Set} → List A → A
head (a :: as) = a
head []        = {!!}
```

Agdaism: Throughout, we explain Agda notation and standard library constructs using notes in the margin. A more comprehensive explanation of Agda notation and primitives can be found in appendix A. It explains, for example, implicit arguments (surrounded by curly braces) and the hierarchy of universes starting with Set.

Agdaism: The notation {!!} denotes a *hole* in an incomplete program.

There is no way to reliably produce an element of the abstract type A out of thin air in case the list turns out to be empty at runtime. In other languages, for example, the hole can be filled by raising an unchecked exception. In Haskell, for example, one can write **undefined**, which means that the evaluation of head [] will crash. This is considered unacceptable in Agda, because the type of the function head does not inform the user that this function can crash. Its type promises to always return an A, and types must be reliable. Consequently, the type of head in the Agda standard library makes it evident that this function is partial:

```
head : {A : Set} → List A → Maybe A
head []        = nothing
head (x :: _) = just x
```

Agdaism: The usual definition of the Maybe data-type with its constructors nothing and just can be found in appendix B.

Similarly, functions that diverge when invoked must also be rejected by Agda's type checker, because the type promises a value that will never be delivered:

```
{-# NON_TERMINATING #-}
bad : ∀ {A : Set} → A
bad = bad
```

Agdaism: The **NON_TERMINATING** pragma disables Agda's termination checker.

In other words: functions in Agda must be defined in such a way that the type checker can verify that they are *total*, so that functions always deliver what their type prescribes.

SECOND, AGDA DISTINGUISHES itself with its support for dependent types[6]—i.e., types that depend on values. Dependent types permit us to reflect in types what we learn when we inspect data. To see this, we review the canonical example of dependently typed programming: vector functions. We define the dependent type of vectors as follows:

[6] Building on a line of pioneering languages such as Alf (L. Magnusson 1994), Lego (Pollack 1995), Epigram (McBride 2004), Cayenne (Augustsson 1998), and others, and competing with other dependently typed languages such as Idris (Brady 2013).

Agdaism: Agda supports mixfix names like _::_: when applied, arguments go where the underscores are.

```
data Vec (A : Set) : ℕ → Set where
  []  :                          Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Vectors are lists, annotated with their length. The argument A of Vec that appears before the colon is called a *parameter* of the data type. In contrast, the argument $\mathbb{N}$ appearing after the colon is called an *index*. The indices reveal some information about the data, as they are *forced* to a specific value by the constructors. In this case, the constructor [] forces the length index to zero and the constructor _::_ forces the index to be one greater than the length index of the tail.

Using the index to our advantage, we can reliably implement the function head on *non-empty* vectors:

Agdaism: Arguments appearing in curly braces (e.g., the two arguments $\{A\ n\}$) are called *implicit* and do not have to be provided at the call site; they are inserted by type inference.

```
head : ∀ {A n} → Vec A (suc n) → A
head (a :: as) = a
```

Unlike for lists, we need not provide a branch for the empty vector []. This is *dependent pattern matching*[7,8] at work. The constructor [] *forces* the index to zero. This contradicts the type signature of head, which specifies that the index is suc n.

[7] Coquand 1992. "Pattern matching with dependent types"

[8] Cockx 2017. "Dependent pattern matching and proof-relevant unification"

Dependent pattern matching also helps out when the index is not yet known. For example:

```
append : ∀ {A n m} → Vec A n → Vec A m → Vec A (n + m)
append []        ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

The type of append intrinsically expresses a property of this function in relation to the length of the vectors. The proof of this property is conducted in collaboration with the type checker, mediated by dependent pattern matching. What is going on? By pattern matching on the first argument of append, we *refine* our knowledge about the index n. We can make this explicit, like so:

```
append : ∀ {A n m} → Vec A n → Vec A m → Vec A (n + m)
append {n = .zero}  []        ys = ys
append {n = .suc k} (x :: xs) ys = x :: (append xs ys)
```

The dot in the pattern for the implicit argument n indicates that the constructor is *forced*. Because n is forced, the type of the vector to be returned can also be refined in both branches. In the first branch $0 + m$ *computes* to m by the (left-recursive) definition of +. Hence,

ys is accepted by Agda as a suitable return value. In the second branch, (suc *k*) + *m* computes to suc (*k* + *m*) and *xs* : Vec *A k*. The types of the constructor _::_ and the recursive call to append make everything fit.

In section 2.2–2.3, we will use dependent types to describe well-typed expressions and their safe interpreters. Throughout this thesis we complement our definitions with notes in the margin explaining features and peculiarities of Agda. The basic syntax is summarized in appendix A, including also the coloring of code. Appendix B summarizes the standard library definitions that the presented code relies on.

2.2 *Extrinsic Verification of Language Implementations*

The interplay between inspecting data and the available type information is a key feature of dependently typed languages that enables *intrinsic verification*—i.e., function definitions that express important properties of the implementation in their type. To show this, we first define an untyped interpreter for a small expression language and verify type-safety *extrinsically*. In section 2.3, we then integrate the type-safety theorem into the definition resulting in an intrinsically typed interpreter.

As an example we will use an interpreter for a tiny, typed expression language with natural numbers *n* and booleans *b*:

$$\begin{array}{rcl} t, u, v & ::= & \mathsf{nat} \mid \mathsf{bool} \\ e & ::= & n \mid b \mid \mathsf{plus}\ e\ e \mid \mathsf{ite}\ e\ e\ e \end{array}$$

An inductively defined relation specifies how expressions are typed as either nat or bool:

Figure 2.1: Typing rules of the simply typed expression language.

▼

$$\frac{}{\vdash n : \mathsf{nat}}\ \text{T-Nat} \qquad \frac{}{\vdash b : \mathsf{bool}}\ \text{T-Bool}$$

$$\frac{\vdash e_1 : \mathsf{nat} \qquad \vdash e_2 : \mathsf{nat}}{\vdash \mathsf{plus}\ e_1\ e_2 : \mathsf{nat}}\ \text{T-Plus}$$

$$\frac{\vdash e_1 : \mathsf{bool} \qquad \vdash e_2 : t \qquad \vdash e_3 : t}{\vdash \mathsf{ite}\ e_1\ e_2\ e_3 : t}\ \text{T-Ite}$$

The embedding Exp of the untyped language into Agda is straightforward. A separate data-type Val represents the canonical forms, or values:

```
data Exp : Set where
  num  : ℕ                    → Exp
  bool : Bool                 → Exp
  plus : Exp → Exp            → Exp
  ite  : Exp → Exp → Exp → Exp

data Val : Set where
  num  : ℕ    → Val
  bool : Bool → Val
```

The (definitional) interpreter eval specifies the dynamic semantics of this expression language:[9]

```
eval : Exp → Maybe Val
eval (num n)                    = just (num n)
eval (bool b)                   = just (bool b)
eval (plus e₁ e₂) with eval e₁ | eval e₂
... | just (num n) | just (num m) = just (num (n + m))
... | _            | _            = nothing
eval (ite c e₁ e₂) with eval c
... | just (bool b)             = if b then eval e₁ else eval e₂
... | just (num n)              = nothing
... | nothing                   = nothing
```

Agdaism: The **with** keyword is used for dependent pattern matching on the result of expressions on the left-hand side (McBride et al. 2004) in an Agda definition. Multiple expressions are separated by a vertical bar. Ellipsis (...) on the next line indicates that we omit the parent pattern.

[9] Reynolds 1998a. "Definitional Interpreters for Higher-Order Programming Languages"

The function eval only returns a value (just ...) for some expressions and returns an error indicator (nothing) otherwise. We say that it is a *partial function*, even though it is encoded as a total one using the type constructor Maybe. It is partial for essentially two reasons.

First, because the input expression can be *ill-typed*, in which case evaluation can get *stuck*.[10] In effect we are giving a semantics for all expressions, including ill-typed ones, even if we are only interested in running well-typed expressions.

[10] For example, the ill-typed terms of form ite (num 4) e_1 e_2 cannot be evaluated, because the condition does not reduce to a boolean.

The second reason is more subtle: even if the input expression e is well-typed at a type a, it is not self-evident that a value returned by eval e is indeed a canonical form of type a. In other words, evaluating a boolean expression may well return a number. Clearly, if that is the case, then that is a bug in eval, because the interpreter violates the type contract between the programmer and the language implementation.

Let us verify that this is not the case by proving that eval is *type safe*:[11]

$$\textsc{Type-Safe}\quad \frac{\vdash e : a}{\exists v.\mathsf{eval}\ e \equiv \mathsf{just}\ v \ \wedge\ \vdash v : a}$$

The theorem says that eval is actually *total* on well-typed inputs: for well-typed expressions it always returns just a value. It also says that the value it returns is typed[12] at the same type as the input expression—i.e., evaluation is *type preserving*.

We can mechanically prove this property in Agda by first embedding the typing relation for expressions, as follows:

```
data Ty : Set where nat bool : Ty
data ⊢_∈_ : Exp → Ty → Set where
  num-wt : ⊢ num n ∈ nat
  bool-wt : ⊢ bool b ∈ bool
  plus-wt : ⊢ e₁ ∈ nat → ⊢ e₂ ∈ nat → ⊢ plus e₁ e₂ ∈ nat
  ite-wt  : ⊢ c ∈ bool → ⊢ e₁ ∈ t → ⊢ e₂ ∈ t → ⊢ ite c e₁ e₂ ∈ t

nf : Val → Exp
nf (num n) = num n
nf (bool b) = bool b
```

We define nf (for "normal form") so that we can reuse the typing relation for expressions also for values via this embedding.

The type safety property of eval is proven inductively. In Agda this inductive proof is given as a recursive function. We show it here in full to exemplify what we call extrinsic verification, but the details of the proof are unimportant.

```
eval-safe : ⊢ e ∈ t → ∃⟨ (λ v → (eval e ≡ just v) × (⊢ (nf v) ∈ t)) ⟩
eval-safe num-wt                          = _ , refl , num-wt
eval-safe bool-wt                         = _ , refl , bool-wt
eval-safe (plus-wt p q) with eval-safe p | eval-safe q
... | num n , eq₁ , num-wt | num m , eq₂ , num-wt
  rewrite eq₁ | eq₂                       = _ , refl , num-wt
eval-safe (ite-wt p q r) with eval-safe p
... | bool true  , eq₁ , bool-wt rewrite eq₁ = eval-safe q
... | bool false , eq₁ , bool-wt rewrite eq₁ = eval-safe r
```

The structure of the proof clearly follows the structure of the computation. Most of the work in writing down the proof is in fact

[11] Technically, this theorem is stronger than mere type-safety, stating also that eval terminates.

[12] We extend the typing relation from expressions to values using the fact that values are a subset of expressions.

Agdaism: The syntax ∃⟨ P ⟩ is an Agda idiom for (proof-relevant) existential quantification over all arguments of the parameterized type P. See appendix A.

Agdaism: The type former _≡_ stands for *propositional equality*, whose one constructor is refl. See appendix B.

Agdaism: The **rewrite** keyword can be used on the left-hand side of a definition to rewrite along a propositional equality proof in the context for the right-hand side. See appendix A.

just replicating the computational structure. In the next section we exploit this similarity and define a version of the interpreter that is both total and intrinsically proves type safety, avoiding both the overhead of a partial definition and of an extrinsic proof.

2.3 *Intrinsic Verification of Language Implementations*

The extrinsic proof eval-safe is adequate as conclusive evidence that the function eval behaves in conformance to a specification. It is a bit long as written here, but this can be remedied using more sophisticated (automation) features of modern proof assistants. What is disappointing, is that even though this useful property holds, it does not buy us anything while we define eval! In a language back-end, we are only ever evaluating well-typed terms. If we can prove that evaluation is total on exactly those terms, then arguably eval ought not be partial at all.

We can do better if we simultaneously implement evaluation *and* prove it type safe:[13]

[13] Augustsson et al. 1999. "An exercise in dependent types: A well-typed interpreter"

Figure 2.2: Augustsson's typed interpreter (without variable binding).

▼

```
⟦_⟧ : Ty → Set
⟦ nat  ⟧ = ℕ
⟦ bool ⟧ = Bool

eval : (e : Exp) → ⊢ e ∈ t → ⟦ t ⟧
eval (num n)     num-wt       = n
eval (bool b)    bool-wt      = b
eval (plus e₁ e₂) (plus-wt p q) = eval e₁ p + eval e₂ q
eval (ite c e₁ e₂) (ite-wt p q r) = if (eval c p) then eval e₁ q else eval e₂ r
```

Agdaism: Variables in signatures that are not explicitly bound—e.g., the expression type t in the signatures of eval—are implicitly universally quantified. See also appendix A.

Such remarkable simplicity on the right-hand side! Unlike the previous eval function, this one requires evidence that its input e is well typed. In return, we can be much more precise about its behavior in the return type—i.e., this type of eval expresses type preservation. It is thus evident that evaluating the condition will return a boolean value and we avoid dealing with type errors altogether. Finally, by simultaneously implementing the function and doing the proof, we avoid repeating the structure that they share.[14]

[14] Wright et al. 1994 already note that the strong type safety theorem that includes type preservation allows a language implementation to omit the representation tags that distinguish values of different types. The typed interpreter from Augustsson makes direct use of this fact and does not use a "tagged union" Val, avoiding wrapping and unwrapping of values.

[15] Benton et al. 2012. "Strongly Typed Term Representations in Coq"

[16] Dybjer 1994. "Inductive Families"

Now we just eliminate the middle man: instead of computing on untyped expressions, superimposing its typing, we compute directly on *intrinsically typed terms*[15]. That is, we define an *inductive family*[16] Exp—in place of the untyped Exp used above—that only represents well-typed expressions:

```
data Exp : Ty → Set where
  num  : ℕ                          → Exp nat
  bool : Bool                       → Exp bool
  plus : Exp nat → Exp nat          → Exp nat
  ite  : Exp bool → Exp t → Exp t → Exp t
```

This inductive family effectively merges the syntax of expressions with the well-typing relation. Using these well-typed expressions as the input of our interpreter simplifies the left-hand side, while retaining all the advantages on the right-hand side:

```
eval : (e : Exp t) → ⟦ t ⟧
eval (num n)     = n
eval (bool b)    = b
eval (plus e₁ e₂) = eval e₁ + eval e₂
eval (ite c e₁ e₂) = if (eval c) then eval e₁ else eval e₂
```

◀

Figure 2.3: Intrinsically typed definitional interpreter.

The function eval directly gives a denotational semantics for well-typed terms.[17] Henceforth, this is what we will refer to as 'intrinsically typed' programming: computing directly on well-typed terms, whose representation contains all the evidence of the type derivation.

[17] McBride 2004. "Epigram: Practical Programming with Dependent Types" (§5.2)

We must say a few more words about the terminology. Our use of 'intrinsic' originates from Reynolds[18,19], who explains that in an intrinsic semantics meaning is assigned to typing judgments rather than program phrases. Reynolds discusses intrinsic denotational semantics in pen-and-paper style and relates them to extrinsic semantics, which also assign meaning to ill-typed programs. The approach has been employed in various proof assistants by a number of authors since, as summarized well by Benton et al.[20].

[18] Reynolds 1998b. "Theories of programming languages"

[19] Reynolds 2000. "The meaning of types from intrinsic to extrinsic semantics"

[20] Benton et al. 2012. "Strongly Typed Term Representations in Coq"

Note that an intrinsically typed representation is significantly different from a 'type annotated' representation of terms, where the type is present, but the derivation (i.e., the evidence) is external if anywhere. Some of the work in Coq uses the terminology 'strong specification', meaning a pair $\{ x : A \mid P\ x \}$ of a value of type a together with a proof that x satisfies the property P : Prop (see, for

example, Swierstra[21]). If we take x to be a program term, and P the type derivation, we obtain something similar in spirit, but technically still different from intrinsically typed terms. The difference arises from the fact that the type derivation may contain more structure/information than the raw term. In principle, an intrinsically typed program can compute based on that additional structure, whereas in Coq a property in the universe Prop has no structure. Coincidentally, such a pair appears to be essentially what Harper et al. have in mind when they write: "From a semantic viewpoint programs are seen as intrinsically typed".[22] We thus admit that our interpretation of the terminology "intrinsically typed" may be more narrow than what one may encounter in the literature throughout the decades. That being said, our interpretation appears consistent with the prevalent use of the terms in the last few years.[23]

[21] Swierstra 2009b. "A Hoare Logic for the State Monad"

[22] Harper et al. 2000. "A type-theoretic interpretation of standard ML"

[23] See, for example, the Agda course on programming Language foundations (Wadler et al. 2020).

Dependent types give us extraordinary expressive power to specify properties of functions in their types. The example shows that such precise types can "make a positive contribution to program construction"[24]. The type preservation theorem embedded in the type of eval not only helps us avoid partiality and prove a theorem, but it also serves as a contract that guides the implementation. When we enforce more invariants, we limit the number of ways in which we can go wrong.[25] Comparing different verification techniques, this approach also appears to hit a sweet spot. We prove the type-safety theorem without putting any work in.

[24] McBride 2004. "Epigram: Practical Programming with Dependent Types"

[25] Erik Meijer, who taught me functional programming, instructs his students to "reduce your brain to the size of a peanut and follow the types".

2.4 *Challenging Feasibility*

The previous section demonstrated the potential of intrinsic verification. In the implementation of eval (figure 2.3) everything went just right. We proved type correctness not just for free, but also gained from expressing the theorem intrinsically in the construction of the program. This is the appeal that we pursue in this line of work.

The big question is whether intrinsic verification with the same appeal remains feasible if we apply it to programming languages with more complicated static semantics and type invariants.

Chapters 3–5 will each challenge the claim that the approach is feasible by adding features to the object language that cannot readily be verified using the state of the art without losing the appeal of the approach. The challenges of the chapters are as follows.

- Chapter 3 presents the challenge of intrinsically verifying type safety of an interpreter for a languages with references.
- Chapter 4 presents the challenge of embedding and interpreting sub-structural languages with linear state in a typesafe manner, focusing on a concurrent language with asynchronous, session-typed communication.
- Chapter 5 faces the challenge of type-correct compilation into a low-level bytecode language where control-flow is represented using jumps and labels. The compiler must ensure that labels are bound exactly once.

Although it is technically possible to give intrinsically typed implementations for these challenges by systematically integrating conventional implementations with conventional extrinsic proofs, the resulting specifications and implementations do not come close to appeal of the interpreter in figure 2.3. Rather than a definition where we appear to get the proof of the safety criterion for free, we find that we need additional manual reasoning to get a definition that the Agda type checker accepts. This defeats the purpose: instead of being guided by a precise type, the language developer is forced to convince the type checker that their definitions fit the type by writing additional inline proofs.

To defend the thesis that it is actually feasible to give perspicuous intrinsically typed implementations for the above challenges, I propose new specification techniques using various (logical) specification languages that are shallowly embedded into Agda. Using these embedded logics we can give more concise definitions that avoid much of the specification effort and manual proof work. Moreover, we show that the embedded logics allow us to systematically recover many familiar *functional abstractions*—e.g., monadic interfaces for the implementation of various effects—with stronger types. This is important, because it enables functional abstractions that not only hide computational boilerplate (e.g., threading state), but also *proof boilerplate* (e.g., the invariant of linear state).

In the subsequent chapters, I present these new ideas and logical languages and apply them to the listed challenges, delivering intrinsically verified implementations that are almost entirely free of proof work.

3 Monotone State

> "Do you wish me a good morning, or mean that it is a good morning whether I want it or not; or that you feel good this morning; or that it is a morning to be good on?"
>
> — Gandalf on semantics,
> from *The Hobbit, or There and Back Again*.

Intrinsically typed definitional interpreters for simply typed languages appear to be an appealing approach to specifying dynamic semantics. They are executable, type safe by construction, and can avoid the tedium and overhead of proving an external type safety theorem. By intrinsically capturing type preservation, we avoid having to deal with partiality, so that these interpreters are also high-level and concise.

For this approach to be practical for the specification and implementation of programming language back-ends, it must scale beyond simply typed programming languages. A prevalent feature of many programming languages is the ability to pass information not as immutable values, but as *references* to shared mutable values that exist in a global *store*. For example, in Standard ML (SML), the builtin operations `ref`, `:=`, and `!` are used to create, to assign to, and to read references, respectively:

```
let x : int ref = ref 18;
x := 42;
(* !x == 42 *)
```

References enable sharing—i.e., the reference `x` can be given away to a function that may modify it, and those modifications will be visible to another function that uses `x`:

The technical material of this chapter was published in a refereed paper. It has been rewritten for the presentation in this thesis.

C. B. Poulsen, A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser (2018). "Intrinsically-typed definitional interpreters for imperative languages". In: *Proceedings of the ACM on Programming Languages* 2.POPL, 16:1–16:34. DOI: 10.1145/3158104.

```
fun increment (i : int ref) = i := (! i) + 1;
increment(x)
(* !x == 43 *)
```

Reference assignment cannot update the value of the reference in arbitrary ways: updates of references must *preserve* the type of the reference. In the above example, the parameter `i` is a reference to a cell containing an integer. Hence, the assignment in the body of the the cell can only update the value to an integer. The assignment `i := true` would be ill typed and hence the program would be rejected by the type checker. Type-preserving update also precludes deallocation of references.[1] We say that such an implementation of state is "monotone". Restricting to monotone state is useful, because it can then be guaranteed by the runtime that dereferencing a shared references is *type safe*—e.g., `!x` for any reference `x` of type `t ref` will always give a value of type `t` and never result in runtime memory or type errors.

[1] Unlike, for example, store allocated memory in system programming languages (e.g., C and C++).

Unfortunately, we find that if we try to define the semantics of a functional language with monotone references using an intrinsically typed interpreter, much of the appeal of the intrinsically typed approach is lost. The definition of the interpreter suffers from the integration of the monotonicity invariant of typed state, requiring explicit reasoning about store types in the interpreter. We refer to this overhead for proving type-safety as "(manual) proof work" in the interpreter.

In this chapter, we consider this problem and show that it can be solved using a shallow embedding of (proof relevant) monotone predicates over store typings. We can abstract over a typed state implementation as an approximation of a state monad in the category of monotone predicates, while also regaining the conciseness of the typed interpreters for simply typed languages.

THE CHAPTER is organized as follows. In section 3.1 we first develop an intrinsically typed definitional interpreter for a simply typed lambda calculus, which will serve as the core functional language to which we will add mutable references. In section 3.2 we then give the traditional typing of the operations for mutable references that we add to the core functional language. We also discuss the store-threading big-step operational semantics for these operations, as well as the typing of the source syntax and of the runtime components

(i.e., values and stores). We then give a fictional monadic interpreter for the operations to explain the goal of this chapter and explain why this interpreter is not directly attainable. In section 3.3, we look at the traditional type-safety proof for the big-step semantics, which informs a direct implementation of a type-safe interpreter that does work. The clarity of this direct implementation suffers from manual proof work. To remedy this, we develop *monotone predicates* in section 3.4. In particular, we define the interface of a monotone state monad in the category of monotone predicates. We show that we can write type-safe programs against this interface without the manual proof work. This leads to a complete intrinsically-typed definitional interpreter for the functional language with mutable references in section 3.6. We end the chapter with a discussion of related work (section 3.7) and a conclusion (section 3.8).

3.1 Typed λ-binding, Typed Environments

A simple language with type-safe monotone state consists of a simply typed functional core with the primitive operations on references on top. In this section, we familiarize ourselves with the definition of the intrinsically typed interpreter for the simply typed lambda calculus (STLC). STLC differs significantly from the expression language in sections 2.2–2.3 because it contains *variable binding*. Variable binding is needed in the core languages in order to construct programs that share mutable references.

The formalization and intrinsically typed interpretation of syntax with binding requires some techniques that have become folklore. We will present an intrinsically typed interpreter that uses *typed de Bruijn indices*[2]. In section 3.3, we will again make use of typed de Bruijn indices to also represent typed store locations.

[2] McBride 2004. "Epigram: Practical Programming with Dependent Types" (§4.2)

The syntax of the simply typed λ-calculus (STLC) with a unit type unit is given by the following grammar of types t and expressions e.[3]

$$\begin{array}{lcl} t, u & ::= & \mathsf{unit} \mid t \longrightarrow u \\ f, e & ::= & \mathsf{tt} \mid x \mid \lambda\ (x : t).e \mid e\ e \end{array}$$

[3] The base types for natural numbers and booleans from the expression language in section 2.2 can be added back to STLC without much ado.

The meta-variable x ranges over a countably infinite set of variable names. The static semantics of the languages is given by a typing relation $\Gamma \vdash e : t$ which, as usual, says that e is typed as t in typing context. The typing context is a *map* from variable names to types. Each entry is called a *typing assumption*. We write $\Gamma[x \mapsto t]$ for updating the map Γ at x with value t, and $\Gamma(x)$ for looking up the value of x. The typing relation is defined as usual:

$$\textsc{T-Unit}\quad \Gamma \vdash \mathsf{tt} : \mathsf{unit} \qquad \textsc{T-Var}\quad \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \textsc{T-Abs}\quad \frac{\Gamma[x \mapsto t] \vdash e : u}{\Gamma \vdash \lambda\ (x : t).e : t \longrightarrow u} \qquad \textsc{T-App}\quad \frac{\Gamma \vdash f : t \longrightarrow u \qquad \Gamma \vdash e : t}{\Gamma \vdash fe : u}$$

▲

Figure 3.1: The typing rules of the simply typed lambda calculus.

The most conventional presentation of the dynamic semantics of STLC is a small-step operational semantics, which uses substitution to specify β-reduction of function applications. We are working towards an interpreter, however, which is essentially a functional version of a substitution-free big-step semantics. To give the big-step evaluation rules in that style, we first need to define the set of values v and environments η:

$$\begin{array}{lcl} v & ::= & \mathsf{tt} \mid \mathsf{closure}\ x\ e\ \eta \\ \eta & ::= & \{x \mapsto v, \ldots\} \end{array}$$

That is, a value is either the unit value (tt) or a function *closure* (closure x e η). The η is the environment captured by the closure, giving values for all the free variables in the body e except the argument x. Environments are maps from variable names to values.

We can now define the substitution-free big-step operational semantics $e, \eta \Downarrow v$, denoting that the open expression e evaluates in environment η to the value v:

Figure 3.2: Substitution-free big-step semantics of STLC.

▼

$$\textsc{E-Unit}\quad tt, \eta \Downarrow tt \qquad \textsc{E-Var}\quad x, \eta \Downarrow \eta(x) \qquad \textsc{E-Fun}\quad (\lambda\ (x : t).e), \eta \Downarrow \mathsf{closure}\ x\ e\ \eta$$

$$\textsc{E-App}\quad \frac{f, \eta \Downarrow \mathsf{closure}\ x\ e'\ \eta' \qquad e, \eta \Downarrow v \qquad e', (\eta'[x \mapsto v]) \Downarrow v'}{(fe), \eta \Downarrow v'}$$

In order to embed this language in Agda we embed the syntax of types as an algebraic data type.

```
data Ty : Set where
  unit  : Ty
  _⟶_ : Ty → Ty → Ty
```

When we embed a language with variable binding in a proof assistant, we have to answer the usual question of how we want to represent variables. Lambda terms have been successfully represented in a nameless manner using de Bruijn indices[4]. That is, variables are referred to by their position in the context, which is in turn determined by the nesting of their binders, thus entirely avoiding α-equivalence and renaming.

The intrinsically typed syntax can follow suit and use bound natural numbers to represent variables, which are typed using the context. It turns out, however, that it is a better idea to do away entirely with untyped syntax and only work with syntax representations that also directly give the evidence of their typing. McBride[5] presents variables represented as anonymous evidence of proof-relevant context membership $t \in \Gamma$:[6]

```
Ctx = List Ty

data _∈_ : A → List A → Set where
  here  : a ∈ (a :: as)
  there : a₁ ∈ as → a₁ ∈ (a₂ :: as)
```

Well-typed expressions are then type- and context-indexed sets:

```
data Exp : Ty → Ctx → Set where
  tt   : Exp unit Γ
  var  : t ∈ Γ → Exp t Γ
  fun  : Exp t (u :: Γ) → Exp (u ⟶ t) Γ
  app  : Exp (t ⟶ u) Γ → Exp t Γ → Exp u Γ
```

The types of the constructors integrate the typing rules into the syntax.

[4] De Bruijn 1972. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem"

[5] McBride 2004. "Epigram: Practical Programming with Dependent Types"

[6] Observe the similarity in the structure of list membership and a natural number defined by constructors zero (approximating here) and suc (approximating there).

The intrinsically typed values ⟦ t ⟧ for a type t are mutually defined together with typed environments Env Γ. We again define values by induction on the types.

Agdaism: The type ⊤ is the standard library's unit type with constructor tt.

```
data Closure (t u : Ty) : Set where
  closure : Exp u (t :: Γ) → Env Γ → Closure t u

⟦_⟧ : Ty → Set
⟦ unit     ⟧ = ⊤
⟦ t ⟶ u ⟧ = Closure t u
```

Environments carry values for each type in the context, and are defined using the standard library type All which represents universal quantification over a list.

Agdaism: Note that Agda is happy to disambiguate constructor names for us using the expected types. We follow Agda standard library practice and use the same constructor names for All as for List.

```
data All (P : A → Set) : List A → Set where
  []  : All P []
  _::_ : P a → All P as → All P (a :: as)

Env : Ctx → Set
Env Γ = All ⟦_⟧ Γ
```

That is, environments give values ⟦ t ⟧ for all types t in the context. We define the lookup function that finds the valuation of a variable in an environment:

```
lookup : Env Γ → t ∈ Γ → ⟦ t ⟧
lookup η x = lookup' η x
  where
    lookup' : All P as → a ∈ as → P a
    lookup' (p :: ps) here     = p
    lookup' (_ :: ps) (there x) = lookup' ps x
```

Now we can define the interpreter of open expressions, if we are willing to temporarily overlook a termination issue:

Agdaism: The notation constr x ← f y is known as an 'irrefutable **with** pattern' where the pattern constr x is written on the same line as the scrutinee f y. See appendix A.

```
{-# TERMINATING #-}
eval : Exp t Γ → Env Γ → ⟦ t ⟧
eval tt       η = tt
eval (var x)  η = lookup η x
eval (fun e)  η = closure e η
eval (app f e) η
  with closure e' η' ← eval f η | v ← eval e η =
  eval e' (v :: η')
```

This interpreter closely resembles the substitution-free big-step rules in figure 3.2. This is a more remarkable fact than it seems. The big-step rules are partial, in the sense that they only explain terminating reductions, but also because there is no guarantee that the term f in an untyped application $f\ e$ indeed reduces to a closure, as required for the rule E-App to apply. The intrinsically typed interpreter on the other hand is only partial in the former sense, because, by the type of f and by type preservation, the pattern match on closure $e'\ \eta'$ is exhaustive. Dependent pattern matching makes this obvious to Agda.

It is not obvious to Agda that this function is terminating, because it is not structurally recursive. The case for function application ends with a recursive call on the function body which is not a sub-expression of the function application. To define this function as a total one in Agda, we can either use fueled interpreters[7,8], or as an interpreter in a delay monad[9,10]. For this particular language we could then still show strong normalization extrinsically (Danielsson 2012), but this is besides the point here. First, because our goal is merely to prove the type safety property of the language, and second, because the complete language with the state operations is no longer strongly normalizing.[11] For now we will stick with the above definition, and we will solve the issue with termination in section 3.6 using the delay monad.

Importantly, the interpreter for a simply typed language with lexical variables preserves the appeal of the approach that we showcased in section 1.2. In the next section we will show that this is not also immediately the case when we add monotone state operations to this functional core.

[7] Siek 2013. "Type Safety in Three Easy Lemmas"

[8] Amin et al. 2017. "Type soundness proofs with definitional interpreters"

[9] Capretta 2005. "General recursion via coinductive types"

[10] Danielsson 2012. "Operational semantics using the partiality monad"

[11] It is possible to recover general recursion using a circular reference, which is in turn possible because store locations can contain closures. This trick is known as *Landin's knot*.

3.2 *Mutable, Monotone State*

We now extend the simply typed functional language from the previous section with the operations for creating and operating on typed references. We extend the grammar as follows:

$$
\begin{array}{rcl}
t, u & ::= & \ldots \mid \mathsf{ref}\ t \\
e & ::= & \ldots \mid \mathsf{ref}\ e \mid e := e \mid !\ e \\
v & ::= & \ldots \mid l
\end{array}
$$

The meta-variable l ranges over a countably infinite set of store locations. The static semantics of primitives for manipulating references is as follows:[12,13]

[12] Harper 1994. "A Simplified Account of Polymorphic References"

[13] Pierce 2002. "Types and programming languages" (§13)

$$\textsc{T-ref}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ t} \qquad \textsc{T-deref}\ \frac{\Gamma \vdash e : \mathsf{ref}\ t}{\Gamma \vdash\ !\, e : t} \qquad \textsc{T-assign}\ \frac{\Gamma \vdash e_1 : \mathsf{ref}\ t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 := e_2 : \mathsf{tt}}$$

That is, references to a mutable value of type t are typed as ref t. The rule for := ensures that updates are monotone: we can only assign a value whose type matches the type of the reference.

The dynamic semantics of a language with monotone state threads a store μ. A *store* is a map from locations l to values and we use the same notation for these maps as for environments. We define the big-step relation $e, \eta, \mu \Downarrow v, \mu'$ to denote that e in environment η with store μ evaluates to value v with store μ'. This is the substitution-free version of the big-step semantics of Harper (1994). The rules for the store operations are:

▲

Figure 3.3: Static semantics of store operations.

Figure 3.4: Big-step semantics of store operations.

▼

$$\textsc{E-ref}\ \frac{e, \eta, \mu \Downarrow v, \mu_1 \qquad l \notin \mathsf{dom}(\mu_1)}{\mathsf{ref}\ e, \eta, \mu \Downarrow l, \mu_1[l \mapsto v]} \qquad \textsc{E-deref}\ \frac{e_1, \eta, \mu \Downarrow l, \mu_1}{!\ e_1, \eta, \mu \Downarrow \mu_1(l), \mu_1} \qquad \textsc{E-asgn}\ \frac{e_1, \eta, \mu \Downarrow l, \mu_1 \qquad e_2, \eta, \mu_1 \Downarrow v, \mu_2}{e_1 := e_2, \eta, \mu \Downarrow \mathsf{tt}, \mu_2[l \mapsto v]}$$

The expression ref e creates a fresh location in the store. Dereferencing $!\ e_1$ evaluates by looking up the value of the location that e_1 evaluates to. And finally, assignment $e_1 := e_2$ updates the existing location that e_1 evaluates to with the value the e_2 evaluates to.

The big-step rules for STLC must also be adapted to thread the store. This is a minor modification and we omit the new rules for brevity.

We now turn to the problem of turning the above operational semantics into an intrinsically typed interpreter. We will briefly look at a fictional implementation that assumes that the threading of the store can be taken care of by a state monad. This definition will clarify what we are after, and we will discuss why this fiction is not directly attainable. The definitions of the intrinsically typed expressions of *only* the store operations and their fictional interpreter amount to the following:

```
data Ty : Set where
  unit : Ty
  ref  : Ty → Ty
Ctx = List Ty
data Exp : Ty → Ctx → Set where
  ref    : Exp t Γ → Exp (ref t) Γ
  !_     : Exp (ref t) Γ → Exp t Γ
  _:=_   : Exp (ref t) Γ → Exp t Γ → Exp unit Γ

⟦_⟧ : Ty → Set
⟦ unit ⟧ = ⊤
⟦ ref a ⟧ = Loc a
```

```
eval : Exp t Γ → Env Γ → State ⟦ t ⟧
eval (ref e) η = do
  v    ← eval e η
  addr ← mkref v
  return addr
eval (! e) η = do
  addr ← eval e η
  get addr
eval (e₁ := e₂) η = do
  addr ← eval e₁ η
  v    ← eval e₂ η
  set addr v
```

▲

Figure 3.5: Typed interpreter for stateful language using a fictional monad State. The do-notation for monadic computations is similar to Haskell's and explained in appendix A.

The problems of implementing the typed interpretation of figure 3.5 are hidden in the innocuous looking state operations. We pretend here that the following total functions can be implemented:

```
mkref : ⟦ t ⟧ → State (Loc t)
get   : Loc t → State ⟦ t ⟧
set   : Loc t → ⟦ t ⟧ → State ⊤
```

But which implementation of the types Loc and State and the state operation can live up to that pretense? Traditional operational semantics for stateful languages use lists of values as stores and integer indices into the list as addresses.[14] Representing addresses by integers cannot suffice, because then one can easily construct addresses that are not backed by cells, or backed by cells that contain values of a different type than what the address type promises.

In a type safe implementation, such *forging* of addresses must be prevented. Moreover, in an *intrinsically* typed implementation such forging must be evidently impossible. To accomplish this, we will take inspiration from the syntactic type safety proof of a big-step operational semantics for ML references, as well as from the representation of typed variables.

In the next section we will first focus on a direct, type-safe implementation of the individual state operations. In section 3.4, we then abstract the implementation of the state into a monad. Finally, in section 3.6, we put together a monadic definitional interpreter for the entire language. There we will also deal with the problem that the extension of STLC with state operations is not strongly (or even weakly) normalizing.

[14] Pierce 2002. "Types and programming languages"

3.3 *Typed Locations, Typed Stores*

The type of intrinsically typed interpreters incorporates the type-preservation theorem for the object-language. To prove the preservation theorem for languages with mutable references one first needs to prove a generalized version that makes precise that state operations are invariably *monotone*. In this section, we review this stronger theorem and use it to give an implementation of an intrinsically typed interpreter that manually threads the state and its invariant.

To state type preservation, we need to define the static semantics of the runtime components. The runtime is typed relative to a *store typing* Σ that maps store locations to types. We define typing judgments for values, environments and stores in figure 3.6. The judgment $\Sigma \vdash v : t$ types the value v as type t relative to store typing Σ. Similarly, $\Sigma \vdash \eta : \Gamma$ denotes that the environment η is typed by the typing context Γ. Finally, $\Sigma \vdash \mu$ expresses that the store μ is typed by the store typing Σ.

$$\textsc{T-tt}\quad \Sigma \vdash \mathsf{tt} : \mathsf{unit}$$

$$\textsc{T-clos}\quad \frac{\Sigma \vdash \eta : \Gamma \qquad \Gamma, (x : t) \vdash e : u}{\Sigma \vdash \mathsf{closure}\ x\ e\ \eta : t \longrightarrow u}$$

$$\textsc{T-loc}\quad \frac{\Sigma(l) = t}{\Sigma \vdash l : \mathsf{ref}\ t}$$

$$\textsc{T-env}\quad \frac{\forall\ (x : t) \in \Gamma.\ \Sigma \vdash \eta(x) : t}{\Sigma \vdash \eta : \Gamma}$$

$$\textsc{T-store}\quad \frac{\forall\ l \in \mathsf{dom}(\mu).\ \Sigma \vdash \mu(l) : \Sigma(l)}{\Sigma \vdash \mu}$$

Figure 3.6: Static semantics of values, environments, and stores.

That is, locations are typed by looking up the type of the associated cell in the store typing Σ. Environments are typed by variable-wise relating the values in the environment with types in a context. Stores are typed by location-wise relating the values in the store μ by the types in the store typing Σ.

The type preservation proof of the state operations depends crucially on monotonicity of the state operations. Monotonicity says that evaluation of expressions can *extend* the store with new, typed cells, but not drop cells or change the type of existing cells.

▲

We can express this concisely using a partial order $\supseteq$ on store types.

$$\textsc{Preservation}\quad \frac{\Gamma \vdash e : t \qquad \Sigma \vdash \eta : \Gamma \qquad \Sigma \vdash \mu \qquad e, \eta, \mu \Downarrow v, \mu'}{\exists\, \Sigma' \supseteq \Sigma \quad \text{s.t.} \qquad \Sigma' \vdash \mu' \qquad \Sigma' \vdash v : t}$$

That is, the evaluation of an expression e *preserves* the type t of the expression and only monotonically updates the store μ to a store μ'. The monotonicity invariant is enforced by the existential quantification over a store typing $\Sigma' \supseteq \Sigma$. Note that we are forced to existentially quantify of Σ' because the exact extension (if any) can in general not be known without evaluating e—that is, the extension is not statically determined. We leave the order itself abstract for now. A sensible choice would be to let $\Sigma' \supseteq \Sigma$ denote that Σ is a 'prefix' of Σ', so that new locations $l + 1, \ldots$ in Σ' will not affect existing locations $0, \ldots, l$ in Σ.

The necessity of monotonicity becomes apparent when we consider, for example, the case for reference assignment in the proof of this theorem, which is carried out by induction on the typing derivation. In that case we have the following premises and goal:

$$\frac{\Gamma \vdash e_1 := e_2 : \mathsf{unit} \qquad \Sigma \vdash \eta : \Gamma \qquad \Sigma \vdash \mu \qquad (e_1 := e_2), \eta, \mu \Downarrow \mathsf{tt}, \mu_2}{\exists\, \Sigma_2 \supseteq \Sigma \quad \text{s.t.} \qquad \Sigma_2 \vdash \mu_2 \qquad \Sigma_2 \vdash \mathsf{tt} : \mathsf{unit}}$$

The third conclusion ($\Sigma_2 \vdash \mathsf{tt} : \mathsf{unit}$) is trivial regardless of Σ_2.

By inversion on the typing of assignment we have $\Gamma \vdash e_1 : \mathsf{ref}\ t$ and $\Gamma \vdash e_2 : t$ for some t. By inversion of the big-step relation we have $e_1, \eta, \mu \Downarrow l, \mu_1$ and $e_2, \eta, \mu_1 \Downarrow v, \mu_2$. From the induction hypothesis, we get that there exists some $\Sigma_1 \supseteq \Sigma$ such that $\Sigma_1 \vdash \mu_1$ and $\Sigma_1 \vdash l : \mathsf{ref}\ t$. The key observation is that value and environment typing is preserved by store extension, so that we can prove $\Sigma_1 \vdash \eta : \Gamma$. We then again appeal to the induction hypothesis and get that there exists some $\Sigma_2 \supseteq \Sigma_1$ such that $\Sigma_2 \vdash \mu_2$ and $\Sigma_2 \vdash v : t$. We again use preservation of value typing under store extension to 'weaken' the typing of l to $\Sigma_2 \vdash l : \mathsf{ref}\ t$. We can now conclude using transitivity of $\supseteq$, and using the fact that updating the store $\mu_2[l \mapsto v]$ preserves the typing Σ_2 of μ_2 because the type $\mathsf{ref}\ t$ of l matches the type t of v under Σ_2.[15]

Working towards an intrinsically typed interpreter we first define store types, typed locations, values, and environments and stores.

◀

Figure 3.7: Type-preservation theorem for type-safe monotone state.

[15] Note that any operations that can *free* or change the type of an existing storage an existing location (i.e., strong update) would break this type safety argument, as the evaluation of e_2 could free or change the type of the location l. We revisit type-safe strong updates in chapter 4.

The representation of typed values can be understood as an embedding of the typing judgment for values. Hence, values are store-type indexed. Again, it works best to do entirely away with untyped syntax. As a result, we end up with a representation for locations that coincides with the representation for variables. Because locations are sometimes understood as the dynamic counterpart to variables this is not unexpected.

The definition of the runtime components—i.e., locations, values, environments, and stores—becomes as follows:

```
StoreTy = List Ty

Loc : Ty → StoreTy → Set
Loc a Σ = a ∈ Σ

data Closure (t u : Ty) : StoreTy → Set where
  closure : Exp u (t :: Γ)
          → Env Γ Σ
          → Closure t u Σ
```

```
⟦_⟧ : Ty → StoreTy → Set
⟦ unit      ⟧ Σ = ⊤
⟦ ref t     ⟧ Σ = Loc t Σ
⟦ t ⟶ u ⟧ Σ = Closure t u Σ

Env : Ctx → StoreTy → Set
Env Γ Σ = All (λ a → ⟦ a ⟧ Σ) Γ

Store : StoreTy → Set
Store Σ = All (λ a → ⟦ a ⟧ Σ) Σ
```

Figure 3.8: Intrinsically typed runtime components. ▲

Intrinsically typed locations are thus evidence that the store type has a location with the given type. The evidence is proof-relevant, specifying exactly where to find this particular location. Because of this, a typed location $l \in \Sigma$ can be looked up in a store Store Σ, just like a typed variable $t \in \Gamma$ can be looked up in an environment Env Γ:

```
get : Store Σ → Loc t Σ → ⟦ t ⟧ Σ
```

The fact that get is total means that typed addresses cannot be forged in a way that type safety is broken! A typed address can always be dereferenced safely in a store that agrees on the store type. In addition to get, we can also define intrinsically type-preserving, total store update set:[16]

```
set : Loc t Σ → ⟦ t ⟧ Σ → Store Σ → Store Σ
set = update
  where
    update : x ∈ xs → P x → All P xs → All P xs
    update (here refl) px′ (px :: μ) = px′ :: μ
    update (there i)   px′ (py :: μ) = py :: update i px′ μ
```

[16] Defined in terms of update, which is also part of the Agda standard library.

Typed dynamic binding is represented completely analogous to typed static binding. So far, so good!

WHAT ABOUT the type of the intrinsically-typed interpreter? The type must reflect that the store may get extended, mirroring the formulation of type preservation for the big-step semantics in figure 3.7:

```
eval : Exp t Γ
     → Env Γ Σ → Store Σ
     → ∃⟨ (λ Σ′ → Σ′ ⊇ Σ × Store Σ′ × ⟦ t ⟧ Σ′) ⟩
```

◀

Figure 3.9: Type of an interpreter for the state operations.

A direct implementation of the interpreter at this type is possible, but tedious. The case for assignment to a reference is illustrative:

```
eval (e₁ := e₂) η μ₁ =
  case eval e₁ η μ₁ of λ
    (_ , ext₁ , μ₂ , loc) →
      case eval e₂ (weaken ext₁ η) μ₂ of λ
        (_ , ext₂ , μ₃ , v) →
          let
            ext = ⊇-trans ext₂ ext₁
            μ₄  = set (weaken ext₂ loc) v μ₃
          in _ , ext , μ₄ , tt
```

◀

Figure 3.10: Direct implementation of reference assignment.

This implementation has exactly the same structure as the preservation proof at the beginning of this section. The recursive invocations of eval with, for example, the store μ_1, results in some updated store μ_2. Importantly, this store is not unrelated to the store μ_1: we also receive evidence ext_1 that it is an extension of μ_1. Mirroring the type preservation proof, this enables us to argue that the environment η that was intrinsically typed with respect to the type of μ_1 is also well-typed with respect to the type of μ_2. We make that argument using the function weaken, mirroring the lemmas that environment typing is preserved by store extension. We use an overloaded function weaken function that *weakens* store-type indexed types P using an extension fact:

```
weaken : Σ₂ ⊇ Σ₁ → P Σ₁ → P Σ₂
```

Agdaism: In Agda this overloading is accomplished using a typeclass. We define a typeclass IsMono that accomplishes this purpose in section 3.4.

Zooming out, we see that the existential quantification in the return type of eval forces us to universally quantify over *any extension*

of a present store of type Σ in the *continuation* of any recursive call. These continuations κ that appear as the body of case analyses on recursive invocations to eval have types of the following form, for some result type X and given Σ:

$$\kappa : \exists \langle\, (\lambda\ \Sigma' \to \Sigma' \supseteq \Sigma \times \mathsf{Store}\ \Sigma' \times [\![\ t\ ]\!]\ \Sigma') \,\rangle \to \mathsf{X}$$

If we curry this type a little bit, the universal quantification becomes more readily apparent:

$$\kappa : \forall\ \Sigma' \to \Sigma' \supseteq \Sigma \to \mathsf{Store}\ \Sigma' \times [\![\ t\ ]\!]\ \Sigma' \to \mathsf{X}$$

Hence, the Σ is used as a *lowerbound* on the type of the store.

THERE ARE TWO problems with the direct definition of the interpreter in figure 3.10.

Compare the direct implementation with the fictional monadic computation from section 3.2 in figure 3.5, or the typed expression interpreter from figure 2.3. The present interpreter is much less attractive to write or read than those prior definitions. In part, this is the case because of the explicit threading of the store. In part it is due to the explicit manipulation of store extensions and the explicit weakening of locations, values, environments, and stores that may be required. This *proof work* is comparable to the pen-and-paper argument for type-safety of reference assignment that we gave in section 3.2. The remainder of this chapter will be about systematically eliminating this manual proof work, in order to obtain a more perspicuous definitional interpreter.

An intrinsically typed definitional interpreter for a stateful language store also ought to be able to abstract over the exact implementation of the store operations. This includes the threading of the store, as well as the necessary weakening of semantic components and values as a result of possible store extension. Monadic interfaces—e.g., MonadStore—have traditionally fulfilled the purpose of abstracting over the exact implementation of effects.[17,18] Unfortunately, neither regular monads for Set/Type (as used in many functional languages), nor parameterized/indexed monads[19] or monads over indexed sets[20] suffice for intrinsically typed, monotone state.

In the next section, we propose a novel solution for solving these problems by systematically developing a shallow embedding of

[17] Moggi 1991. "Notions of computation and monads"

[18] Wadler 1992. "The Essence of Functional Programming"

[19] Atkey 2009. "Parameterised notions of computation"

[20] McBride 2011. "Kleisli Arrows of Outrageous Fortune"

constructions in the category of *monotone predicates*. Like the continuations of the presented interpreter, functions in the category of monotone predicates universally quantify over store extensions. This enables us to develop a familiar looking monadic interface that abstracts the implementation of mutable store operations. In section 3.6 we then show that the store threading of figure 3.9 indeed implements this abstract monadic interface.

3.4 *Proof Relevant, Monotone Predicates*

We define proof-relevant *monotone predicates* and show that a *strong monad* in this category can be used to abstract the interface of monotone state. We also show that we can write monadic interpreters in Agda using this abstraction that regain the appeal that we lost in the direct implementation of the interpreter of the previous section.

We will work abstractly with (proof-relevant) predicates on some set $\mathcal{R}$ with a partial order $\_\geq\_$. We will later instantiate $\mathcal{R}$ to StoreTy.

We take monotone predicates to be predicates in Set that are then proven monotone in an ad-hoc manner using a typeclass IsMono:

```
Pred : Set → Set₁
Pred R = R → Set

record IsMono (P : R → Set) : Set where
  field
    weaken : j ≥ i → P i → P j
```

As in the previous section, we will use weaken as a overloaded function, whose implementation is given by instance definitions of this IsMono typeclass (not shown). Locations, values and environments are all monotone predicates.

We adopt the notation of the Agda standard library for predicates:

```
_⇒_ : Pred R → Pred R → Pred R
P ⇒ Q = λ r → P r → Q r

_∩_ : Pred R → Pred R → Pred R
P ∩ Q = λ r → P r × Q r

∀[_] : Pred R → Set
∀[ P ] = ∀ {r} → P r
```

That is, we write $\Rightarrow$ (respectively $\cap$) for the pointwise lifting of functions (respectively products) from Set to Pred $\mathcal{R}$. We can embed predicates into Set again using the universal closure $\forall[\_]$.

A special role is reserved for functions that quantify over all extensions of some given lowerbound. We define a record of such functions $P \Rrightarrow Q$ that denote such *Kripke* functions[21,22]:

```
record _⇛_ (P Q : Pred ℛ) (i : ℛ) : Set where
  constructor kripke
  field
    apply : ∀ {j} → j ≥ i → P j → Q j
```

[21] Kripke 1963. "Semantical Considerations on Modal Logic"

[22] Abel 2013. "Normalization by Evaluation: Dependent Types and Impredicativity"

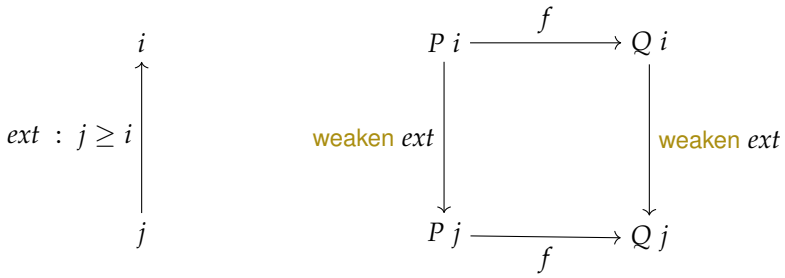
Because we use a record, Kripke functions are introduced using the constructor kripke of the record with the fields as its parameters (i.e., kripke : $(\forall j \to j \geq i \to P\ j \to Q\ j) \to (P \Rrightarrow Q)\ i$). Unlike pointwise lifted functions $P \Rightarrow Q$, Kripke functions are themselves monotone predicates:

```
instance
  ⇛-mono : IsMono (P ⇛ Q)
  IsMono.weaken ⇛-mono ext₁ f =
    kripke (λ ext₂ px → apply f (≥-trans ext₁ ext₂) px)
```

To a category theorist, monotone predicates are an approximation of (contravariant) functors from a partial ordered set $\mathcal{R}$ to Set. Such functors commonly go by the name *presheaves*. The functorial action of such functors on morphisms in $\mathcal{R}$ corresponds to the operation weaken of the IsMono typeclass. The morphisms of a category of functors are natural transformations. An Agda function f : ∀[$P \Rightarrow Q$] between two monotone predicates P and Q should morally correspond to the component of such a natural transformation. This is the case when the application of f commutes with weakening, as follows:

Figure 3.11: Naturality of the function f between two monotone predicates P and Q.

▼



Kripke functions approximate the exponents of this category. We only approximate the categorical notions, because we do not enforce that the functor laws are satisfied, nor that morphisms are natural. This approximation suffices for proving type safety of interpreters. The laws become relevant if one wants to reason about typed interpreters.

In the category of sets, computations that yield values of type B in the presence of computational effects are conventionally represented as computations $M\ B$ in some monad M.[23,24] Such computations can be constructed by sequential composition of effectful primitives using the monadic bind m >>= f, where $m : M\ A$, and f is a potentially effectful continuation $A \to M\ B$. In order to abstract effectful computations such as monotone state, we define the interface of (an approximation of) a monad in the category of monotone predicates. We use Kripke functions as continuations, inspired by our observation about the continuations of the direct implementation in figure 3.10.

[23] Moggi 1991. "Notions of computation and monads"

[24] Wadler 1998. "The Marriage of Effects and Monads"

```
record IsMPMonad (M : Pred R → Pred R) : Set₁ where
  field
    return : ∀[ P ⇒ M P ]
    _≥=_ : ∀[ M P ⇒ (P ⇛ M Q) ⇒ M Q ]
```

To abstract from the specific implementation of monotone state, we define the interface MonadStore $L\ V\ M$ of a monad M that implements the primitive operations of monotone state locations L, and values V, both intrinsically typed by types T.

Figure 3.12: The interface of a monad that implements the primitives for references.

▼

```
record MonadStore {T} (L : T → Pred R) (V : T → Pred R) M : Set₁ where
  field
    {{monad}} : IsMPMonad M
    mkref     : ∀[ V t ⇒ M (L t) ]
    get       : ∀[ L t ⇒ M (V t) ]
    set       : ∀[ L t ⇒ V t ⇒ M (λ _ → ⊤) ]
```

In section 3.6 we will show that the threading of the store of the interpreter in figure 3.9 can be encapsulated by this interface, but let us first examine the status of a simpler example inspired by the thesis of Swierstra[25]:

[25] Swierstra 2009a. "A functional specification of effects" (§6.2)

```
data T : Set where
  NAT : T

Nums : T → ℛ → Set
Nums NAT Σ = ℕ

copy-inc : ∀ {M} {{_ : MonadStore L Nums M}}
         → ∀[ L NAT ⇒ M (λ _ → ⊤) ]
copy-inc r =
  get r ≥= kripke λ ext₁ n →
  (if (n ≤? 10)
    then (mkref n ≥= kripke λ _ _ → return tt)
    else (return tt)) ≥= kripke λ ext₂ m →
  set (weaken (≥-trans ext₁ ext₂) r) (n + 1)
```

▲

Figure 3.13: A simple program with statically unknown effects on the type of the store and explicit weakening.

We define the function copy-inc for any monad M that implements the interface MonadStore, using any monotone predicate L denoting T-indexed references to T-indexed Nums. The function takes a location $r : L$ Nat and gets its value n. It then checks if the value is less than or equal to 10, and if that is the case it allocates another reference, forgets about it and yields tt. If n is greater than 10, it just yields tt immediately. Finally, it increments the value behind the location l.

This function is a simple example of a type-safe, stateful function, whose effect on the shape of the store is not statically determined. We see that we can write such functions in a type-safe manner against a monotone predicate monad because each continuation of a bind universally quantifies over extended stores. At the same time, by enforcing store extension, we can safely use the location l after computations whose precise effect is not statically known, *provided* that we weaken the location using the extension facts passed to the continuations.

In the next section we show that we can systematically avoid manual weakening by using a slightly different monad interface that is equivalent in terms of expressive power.

3.5 *Programming with a Strong Monad for State*

The monotone predicate monad bind ≥= is not arbitrary. It approximates the *internal* bind of a monad in the category of monotone predicates.

That is, a bind morphism:

```
bind : ∀[ (P ⇛ M Q) ⇒ (M P ⇛ M Q) ]
```

When we unfold the definition of the Kripke function, we obtain:

```
bind : ∀ {i j} → j ≥ i
     → (∀ {k} → k ≥ j → P k → M Q k)
     → M P j → M Q j
```

McBride[26] explains how the two universal quantifiers of such a bind are of different 'polarity'. While we (i.e., the player) choose i and j, it is the monad (i.e., the opponent) who picks k. Because we are free to choose both i and j—and because the choice of i is inconsequential—we can be pragmatic and choose $i = j$, so that by reflexivity of the order the bind can be simplified to `_≥=_` without loss of generality.

The internal bind is characteristic for a *strong monad*. In the semantics of effectful computations, the fact that a monad is strong has been traditionally important.[27] It has played a lesser role in programming, however, because every monad is strong in the category of sets. Because we depart from the category of sets, we have to take special care.

[26] McBride 2011. "Kleisli Arrows of Outrageous Fortune"

[27] Moggi 1991. "Notions of computation and monads"

Once we understand how our effectful computations can be defined using a strong monad in the category of monotone predicates,[28] we can use the fact that the internal bind is known to be equivalent to the *external bind* `_>>=_` (mapping morphisms of the category) and an operation `_&_` called *tensorial* strength:

```
_>>=_ : M P i → ∀[ P ⇒ M Q ] → M Q i
m >>= f = m ≥= kripke λ ext p → f p

_>>_ : M P i → ∀[ M Q ] → M Q i
p >> q = p >>= λ _ → q

_&_ : {{mono : IsMono Q}} → ∀[ Q ⇒ M P ⇒ M (Q ∩ P) ]
q & m = m ≥= kripke λ ext p → return (weaken ext q , p)
```

As for monads in the category of sets, the second argument of the external bind (`∀[ P ⇒ M Q ]`) is just a unary function. We define the usual syntactic sugar `_>>_` for sequential composition of an effectful computation that ignores the result of the preceding computation. Tensorial strength absorbs values of a monotone predicate Q from the context into a monadic computation.

[28] With special thanks to a friendly anonymous POPL reviewer.

◀

Figure 3.14: The external bind, sequential composition, and tensorial strength for a monad M in the category of monotone predicates.

What do we gain from this perspective? Using tensorial strength we can write intrinsically typed program with statically unknown effects on the typed store *without manual weakening*. Because the continuation of _>>=_ (the second argument) has its usual arity, we can even use Agda's builtin do notation now. We show the desugaring of the do-notation on the right.

Figure 3.15: A monadic program using do-notation and tensorial strength (left), and its desugaring (right).

▼

```
copy-inc : ∀ {M} {{_ : MonadStore L Nums M}}
  → ∀[ L NAT ⇒ M (λ _ → ⊤) ]
copy-inc r₁ = do
  r₂ , n ← r₁ & get r₁
  r₃ , tt ← r₂ & (if (n ≤? 10)
    then (do mkref n; return tt)
    else return tt)
  set r₃ (n + 1)
```

```
copy-inc : ∀ {M} {{_ : MonadStore L Nums M}}
  → ∀[ L NAT ⇒ M (λ _ → ⊤) ]
copy-inc r₁ =
  (r₁ & get r₁) >>= λ (r₂ , n) →
  (r₂ & if (n ≤? 10)
    then (mkref n >> return tt)
    else return tt) >>= λ (r₃ , tt) →
  set r₃ (n + 1)
```

3.6 *A Complete Monadic Definitional Interpreter for References*

In this section we put all the ingredients together and define an interpreter for the composition of the core functional language of section 3.1 and the state operations of section 3.3. We will see that we obtain an interpreter that very closely resembles the fictional interpreter in figure 3.5. We begin by summarizing the typed syntax:

```
data Exp : Ty → Ctx → Set where
  tt    : Exp unit Γ
  var   : t ∈ Γ → Exp t Γ
  fun   : Exp t (u :: Γ) → Exp (u ⟶ t) Γ
  app   : Exp (t ⟶ u) Γ → Exp t Γ → Exp u Γ
  ref   : Exp t Γ → Exp (ref t) Γ
  !_    : Exp (ref t) Γ → Exp t Γ
  _:=_  : Exp (ref t) Γ → Exp t Γ → Exp unit Γ
```

We can interpret the language in a monad transformer stack consisting of the reader monad for passing the environment, wrapping the state monad for threading the state. This combination is again a monad and threads the store in the same way as the direct implementation in figure 3.10:

```
M : Ctx → Pred StoreTy → Pred StoreTy
M Γ P = Env Γ ⇒ Store
      ⇒ (λ Σ → ∃⟨ (λ Σ′ → Σ′ ⊇ Σ × Store Σ′ × P Σ′) ⟩)
```

The type M implements the interface of a monotone predicate monad. We include the definition for the curious readers:

```
instance state-mpmonad : IsMPMonad (M Γ)
IsMPMonad.return state-mpmonad px η μ = -, ⊇-refl , μ , px
IsMPMonad._≥=_ state-mpmonad m f η μ₁ =
  case m η μ₁ of λ
    (_ , ext₂ , μ₂ , px) →
      case apply f ext₂ px (weaken ext₂ η) μ₂ of λ
        (_ , ext₃ , μ₃ , qx) → _ , ⊇-trans ext₂ ext₃ , μ₃ , qx
```

The monadic operations exactly encapsulate the tedious work of the direct interpreter (figure 3.10). By also passing the environment in the monad, the bind also takes care of weakening the environment.

We can leave the order relation abstract to clients of the monad and to the implementation of the MPMonad interface, which only depended on the relation being reflexive and transitive. The choice of the order is relevant, however, to the implementation of the MonadStore operations. We will implement mkref as appending a new cell to the end of the store, so that existing locations are morally unaffected[29]. This means that we should implement store extension $\Sigma' \supseteq \Sigma$ as witnessing that Σ is a prefix of Σ':

[29] Their index changes, but the term remains the same. This means that in principle the weakening of locations is computationally irrelevant. We have not attempted to convince Agda of this.

```
data _⊇_ : List A → List A → Set where
  here  : as ⊇ []
  there : as ⊇ bs → (a :: as) ⊇ (a :: bs)
```

This relation is reflexive (⊇-refl) and transitive (⊇-trans).

The details of the implementation of the MonadStore interface are not very important. Nonetheless, we give the implementation of the state operations based on basic operations of heterogeneous lists for the sake of completeness in figure 3.16.

```
mkref : ∀[ ⟦ t ⟧ ⇒ M Γ (Loc t) ]
mkref v η μ =
  let ext = ⊇-++ ⊇-refl
  in _ , ext , ++⁺ (weaken ext μ) (weaken ext v :: []) , ++⁺ʳ (here refl)

get  : ∀[ Loc t ⇒ M Γ ⟦ t ⟧ ]
get ℓ η μ  = _ , ⊇-refl , μ , lookup μ ℓ

set  : ∀[ Loc t ⇒ ⟦ t ⟧ ⇒ M Γ (λ _ → ⊤) ]
set ℓ v η μ  = _ , ⊇-refl , μ [ ℓ ]:= v , tt
```

```
⊇-++  : xs ⊇ ys → (xs ++ zs) ⊇ ys
++⁺   : All P xs → All P ys
      → All P (xs ++ ys)
++⁺ʳ  : x ∈ ys → x ∈ (xs ++ ys)
_[_]:=_ : All P xs → x ∈ xs → P x
      → All P xs
```

▲

Figure 3.16: Implementation of the MonadStore interface. The types of auxiliary functions (primarily from the Agda standard library) are given on the right.

In addition to the state interface, the type M also implements the operations of a reader monad. In particular, we will make use of the operation ask for retrieving the environment, and local for modifying the environment:

```
ask : ∀[ M Γ (Env Γ) ]
ask η μ = _ , ⊇-refl , μ , η

local : ∀[ (Env Γ ⇒ Env Δ) ⇒ M Δ P ⇒ M Γ P ]
local f m η μ = m (f η) μ
```

The implementation of the intrinsically typed definitional interpreter is now straightforward, still temporarily postponing the issue of non-termination:

Figure 3.17: Complete intrinsically typed interpreter.

▼

```
{-# NON_TERMINATING #-}
eval : Exp t Γ → ∀[ M Γ ⟦ t ⟧ ]
eval tt = return tt
eval (var x) = do
  η ← ask
  return (lookup η x)
eval (fun e) = do
  η ← ask
  return (closure e η)
eval (app f e) = do
  closure body η′ ← eval f
  η″ , v          ← η′ & eval e
  local (λ _ → v :: η″) (eval body)
eval (ref e) = do
  v ← eval e
  mkref v
eval (! e) = do
  ℓ ← eval e
  get ℓ
eval (r := e) = do
  ℓ      ← eval r
  ℓ′ , v ← ℓ & eval e
  set ℓ′ v
```

In this definition we only have to use tensorial strength twice: once for every binary expression in the language. Compare this interpreter with the direct implementation in figure 3.10! The present

intrinsically typed definitional interpreter is completely free of manual proof work, yielding a definition which is almost identical to the fictional definition in figure 3.5.

As promised, we can deal with the problem that the language is not weakly normalizing using existing techniques. For example, we can modify M to use the sized delay monad[30] from the Agda standard library, as follows:

[30] Abel et al. 2014. "Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types"

```
M : Size → Ctx → Pred StoreTy → Pred StoreTy
M i Γ P = Env Γ ⇒ Store
        ⇒ (λ Σ → Delay ∃⟨ (λ Σ′ → Σ′ ⊇ Σ × Store Σ′ × P Σ′) ⟩ i)
```

The index i is essentially a lower bound on observation depth. It is out of the scope of this thesis to discuss in detail how the sized delay monad works. It is discussed in detail by Abel et al.[30]. Incorporating the Delay monad has very little impact on the definition of the interpreter. The only difference is that recursive calls to eval in the version with Delay are replaced by an auxiliary function ▶eval that is defined as a *guarded call* to eval. The resulting interpreter passes Agda's totality checker.

3.7 *Related Work*

In this chapter we showed that a shallow embedding of monotone predicates can be used as a meta-language to systematically and intrinsically explain the invariant of type-safe references. We introduced monads in the category of monotone predicates to abstract an intrinsically typed implementation of type-safe state.

We are not the first to investigate monadic abstractions for intrinsically typed effectful programs. In particular, we want to contrast monotone predicate monads with predicate monads (i.e., regular monads over indexed sets)[31] and parameterized monads[32] (or, indexed monads over regular sets) by comparing their characterizing binds. In figure 3.18, we unfold all type-level abstractions and are very explicit about where the variables are quantified to make the differences more distinct.

[31] McBride 2011. "Kleisli Arrows of Outrageous Fortune"

[32] Atkey 2009. "Parameterised notions of computation"

```
– Bind of a parameterized monad M
_>>=ᵢ_ : ∀ {M : R → R → Set → Set} {i j k}
       → M i j A → (A → M j k B) → M i k B

– Internal bind of a predicate monad M,
– and external bind of a monotone predicate monad M.
_>>=_ : ∀ {M : Pred R → Pred R} {P Q} {i}
      → M P i → (∀ {j} → P j → M Q j) → M Q i

– Internal bind of a monotone predicate monad M.
_≥=_ : ∀ {M : Pred R → Pred R} {P Q} {i}
     → M P i → (∀ {j} → j ≥ i → P j → M Q j) → M Q i
```

◀

Figure 3.18: Comparison of the binds of various monads.

The subtle differences in the placement of the indices i, j, k make all the difference. In the bind of the parameterized monad we universally quantify over the indices at the outside. This means that we have to give all three *upfront* when we call the bind. This is appropriate for composing state computations whose effect on the store type is statically known. Contrast this with the bind of a predicate monad: the quantification over the index j of the continuation happens on the inside. This means that we do not have to give it upfront, but after running the computation $M\ P\ i$ we continue in a store typed j which is statically unrelated to the store type i. Finally, the bind of a monotone predicate monad also quantifies over j on the inside of the continuation but relates it to the outer index i by an extension fact $j \geq i$.

McBride (2011) already showed that predicate monads generalize parameterized monads. Monotone predicates monads again generalize predicate monads. The latter can be reconstructed from the former by instantiating the partial order with the full relation.

The differences between these monads are significant to the application of interpreting mutable references. Swierstra[33] investigates the application of a parameterized IO monad to mutable references in chapter 6 of his thesis. Because this monad is indexed by both the type of the heap before *and* after an effectful computation, it is only applicable to programs whose effect is statically determined. This precludes both the example in figure 3.13 and the interpretation of ML-like mutable references. Predicate monads are also not readily suitable for these programs. Because the store type j in

[33] Swierstra 2009a. "A functional specification of effects"

the continuation of the bind is unrelated to initial store type i, we cannot guarantee that previously obtained store locations can still be dereferenced safely.

Swierstra also observes the problem of requiring manual weakening in the resulting programs and proposes a partial solution based on *smart* operations that take a store extension witness as an instance argument. These arguments are filled in by a decision procedure for $\supseteq$ whenever the store types are concrete. As he also observes, this does not work when the store types contain variables, as in the example in figure 3.13. Monadic programs that use tensorial strength solve this issue with little syntactic overhead (section 3.4), even if the exact effect of an effectful operation is not statically known.

THE MONOTONE STATE MONAD approximates a monad in the category of contravariant presheaves over a partially ordered set. Plotkin and Power describe a 'local state monad' as a construction in this category.[34] The implementation of the MonadStore interface in section 3.6 approximates their categorical definition. They remark that the monad is indeed a lawful strong monad.

DEFINITIONAL INTERPRETERS date back to the earliest days of computer science, and there was already a large body of literature by the time Reynolds[35] published his seminal study. Definitional interpreters have been somewhat overshadowed, particularly when proving type safety is a primary goal, by small-step operational semantics[36,37,38]. However, recent years have seen a revival in interest in functional encodings of semantics. For example, Danielsson[39] shows how to prove type safety for definitional interpreters for languages with non-termination using the partiality monad[40]. A functional definition of dynamic semantics requires no separate proof of determinism. The semantics is also readily executable, and thus does not require a separate functional definition that then needs an adequacy result.[41]

THERE HAS ALSO been a lot of interest in using dependent types to represent strongly typed terms for programming languages. The approach of intrinsically typing terms and operations has been applied to System F[42], as well as dependently typed languages[43]. The connection between intrinsically typed syntax and definitional interpreters has also been made early and often. For example, early

[34] Plotkin et al. 2002. "Notions of Computation Determine Monads"

[35] Reynolds 1972. "Definitional interpreters for higher-order programming languages"

[36] Wright et al. 1994. "A Syntactic Approach to Type Soundness"

[37] Milner et al. 1997. "The Definition of Standard ML, Revised"

[38] Pierce 2002. "Types and programming languages"

[39] Danielsson 2012. "Operational semantics using the partiality monad"

[40] Capretta 2005. "General recursion via coinductive types"

[41] Amin et al. 2017. "Type soundness proofs with definitional interpreters"

[42] Chapman et al. 2019. "System F in Agda, for Fun and Profit"

[43] Chapman 2009. "Type Theory Should Eat Itself"

work on generalized algebraic datatypes[44,45] often uses definitional interpreters as a motivational example.

The relevance of 'Kripke functions' that close over extensions is not surprising. Kripke semantics have seen an abundance of applications in semantics for (manipulations of) terms over contexts and for semantic models of type systems with references[46,47] under the umbrella of 'possible-world semantics'. The connection with Kripke semantics is slightly hidden, but becomes apparent if we factorize the exponent of the category of monotone predicates as follows:

$$\Box\_ : \mathsf{Pred}\ \mathcal{R} \to \mathsf{Pred}\ \mathcal{R}$$
$$\Box\ P = \lambda\ i \to \forall\ \{j\} \to i \leq j \to P\ j$$

$$\_\Rrightarrow\_ : \mathsf{Pred}\ \mathcal{R} \to \mathsf{Pred}\ \mathcal{R} \to \mathsf{Pred}\ \mathcal{R}$$
$$P \Rrightarrow Q = \Box\ (P \Rightarrow Q)$$

where $\Box\_$ is the shallow embedding of the Kripke semantics of the necessity modality of modal logic, upwards closing a predicate over all 'future worlds' j.[48] This exact embedding was used by Allais et al.[49] to define an abstract semantics for λ-terms in an intrinsically scope-and-type safe manner in Agda. In their work, $\mathcal{R}$ is instantiated to typing contexts of λ-terms, and the order on contexts is taken to be a context 'thinning'. Predicates that respect thinnings are called thinnable; a notion that coincides with our definition of 'being monotone'. In later work this was complemented with an embedding of a description language (i.e., a 'universe') for syntaxes with lambda binding.[50] This enables generic proofs of being thinnable for interpretations of descriptions, as well as datatype generic definitions of intrinsically typed traversals that are safe with respect to the scope and type of λ-binding.

Our treatment of the invariants of state have been syntactic, in the sense that safety follows from a type discipline, directly informed by the traditional progress and preservation proof[51]. This is in contrast to work in the spirit of Ahmed[47] and Appel et al.[46], where the type of references is given a semantics as a (step-indexed[52,53]) predicate on the state. Similarly, there has also been work on programming with state monads where computations are indexed with pre- and post-conditions that are formulated as predicates on the state.[54,55] This semantic approach scales to sizable languages such

[44] Bird et al. 1998. "Nested Datatypes"

[45] Altenkirch et al. 1999. "Monadic Presentations of Lambda Terms Using Generalized Inductive Types"

[46] Appel et al. 2007. "A very modal model of a modern, major, general type system"

[47] Ahmed 2004. "Semantics of types for mutable state"

[48] Kripke 1963. "Semantical Considerations on Modal Logic"

[49] Allais et al. 2017. "Type-and-scope safe programs and their proofs"

[50] Allais et al. 2018. "A type and scope safe universe of syntaxes with binding: their semantics and proofs"

[51] Harper 1994. "A Simplified Account of Polymorphic References"

[52] Timany et al. 2017. "Logical relations in Iris"

[53] Timany 2018. "Contributions in Programming Languages Theory: Logical Relations and Type Theory"

[54] Swamy et al. 2013. "Verifying higher-order programs with the Dijkstra monad"

[55] Swierstra et al. 2019. "A predicate transformer semantics for effects (functional pearl)"

as Rust[56] and a polymorphic language with session-typed communication across concurrent processes[57]. The additional expressiveness of the approach comes at the cost of demanding more powerful reasoning principles, which is conducted extrinsically to the definition of the semantics. In proof assistants like Coq and F*, some of the costs of extrinsic proofs can be mediated by automation via tactics or external automatic provers.

[56] Jung et al. 2018a. "RustBelt: Securing the foundations of the Rust programming language"

[57] Hinrichsen et al. 2021. "Machine-checked semantic session typing"

3.8 Conclusion

Augustsson et al.[58] taught us how elegant a type-safe specification of dynamic semantics can be if we define it using an intrinsically typed definitional interpreter in a dependently typed meta-language. A direct implementation of an intrinsically typed interpreter for a language with type-safe references, however, results in a much less elegant specification, obscured by manual proof work (section 3.3). This is disappointing, because state is an effect that is prevalent in many programming languages.

[58] Augustsson et al. 1999. "An exercise in dependent types: A well-typed interpreter"

In this chapter, we showed how to regain a high-level, concise specification using a monadic interpreter (section 3.6). Our key idea is to use a strong monad in the category of monotone predicates (section 3.4) to thread an intrinsically typed store.

Our intrinsically-typed monadic definition demonstrates that intrinsically typed interpreters can scale to type-safe *effectful* languages. The definition of the monadic interface combines ideas from a conventional syntactic progress-and-preservation proof for type-safe references with ideas from the semantics of effectful languages. Exploration and experimentation is necessary to find points in this design space that work well in current day dependently typed programming languages. By staying close to existing functional programming devices, we can lean on existing language support for them (e.g., do-notation) and minimizing the need for additional language support to deal with the remaining proof overhead.

In that vein, an interesting bit of future work would be to explore meta-language support for programming with strong monads. It appears, for example, that Agda could automatically insert tensorial strength in programs that use do-notation by static analysis of the indexed values used in the continuation of the bind. This would be similar to the desugaring of programs written in arrow notation in Haskell.[59]

[59] Paterson 2001. "A new notation for arrows"

4 Linear, Session-Typed Communication

> "And when that work is done, I shall join my warriors and make the final ascension to full mechanisation!"
>
> — Lonely cyberman, Doctor Who

Type safety of monotone state is about ensuring that a resource (i.e., the store) is maintained and remains available to clients (i.e., references). Another kind of resource is one that is *consumed*, so that it ceases to be available after one or multiple uses. The verification of safety properties for languages with consumable resources differs from the verification for monotone invariants, requiring bookkeeping of resource consumption. This is a challenge for intrinsic verification, because we want to avoid the manual bookkeeping.

In this chapter, we will develop a framework for concise, intrinsic specification of typed data, abstracting over the bookkeeping of resource consumption. To accomplish this, we rely on the ideas from the previous chapter and use proof-relevant predicates $\mathcal{R} \to \mathsf{Set}$ to systematically hide the consumption of a resource $\mathcal{R}$. In order to hide the bookkeeping, we develop a very general algebra for *resource composition*. This yields a shallowly embedded separation logic on data that may consume resources.

To demonstrate that this is useful and practical, we develop all these abstractions in the context of concrete programming languages with a consumable resource. In particular, we deliver an intrinsically typed interpreter for a concurrent language with session-typed communication channels between threads. Session types describe communication protocols that the threads must adhere to when they operate on a channel reference. This means that channel references as typed at any point in time will not persist, but are 'used' when threads send or receive and the protocol progresses.

The technical material of this chapter was published in A. Rouvoet, C. Bach Poulsen, R. Krebbers, and E. Visser (2020b). "Intrinsically-typed definitional interpreters for linear, session-typed languages". In: *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 284–298. DOI: 10.1145/3372885.3373818.

In section 4.1, we describe this language and the usual typing of the expression language. We then explain the representation of intrinsically, linearly typed syntax of the expression language in Agda, as well as the typing of the linear runtime in section 4.2. In section 4.3, we show that without appropriate abstractions, defining typed functions over the syntax is unwieldy, and cannot attain the appeal of intrinsically typed definitions demonstrated in chapter 2. We introduce the separation logic in section 4.4 and show that it can be used to give a concise interpreters for a lambda calculus with *linear* variable binding in section 4.5. In section 4.6–4.9, we use the same logic instantiated with a different resource to handle linear (channel) references. We end the chapter with a discussion of the related work and a conclusion.

4.1 Linear, Session-Typed Languages

As a simple session-typed language[1,2] we consider a linearly-typed lambda calculus, extended with primitives for concurrency and communication.

$$
\begin{array}{rcl}
\alpha & ::= & t \ ?\ \alpha \mid t \ !\ \alpha \mid \mathsf{end} \\
t, u & ::= & \mathsf{unit} \mid t \multimap u \mid \alpha \\
f, e & ::= & \ldots \mid \mathsf{fork}\ e \mid \mathsf{mkchan}\ \alpha \mid \mathsf{send}\ e\ e \mid \mathsf{recv}\ e
\end{array}
$$

We adopt the convention to use lowercase greek letters for session types. The session-type t ? α denotes a communication protocol that receives a t and continues as α. Conversely, t ! α sends a t and continues as α. A new thread is created with the expression fork e, where e should be a linear function with a unit argument representing the computation that will be executed on the new thread. The expression send e_1 e_2 sends the result of e_1 on the channel e_2. The expression recv e receives a value from the channel e. The following sample program (in SML inspired pseudo concrete syntax) exchanges a constant number between two threads:

```
let l , r = mkchan (nat ? end);
fork (fun _ => let x, l' = recv l in close l');
let r' = send 42 r;
close r'
```

The program spawns a new channel using `mkchan` with a communication protocol specified by the given session type. This produces

[1] Gay et al. 2010. "Linear type theory for asynchronous session types"

[2] Wadler 2014. "Propositions as sessions"

◀

Figure 4.1: A program with session-typed communication in a fictional language.

two endpoints `l` and `r` of dual session types `nat ? end` and `nat ! end`, respectively. The first type denotes that the endpoint expects to receive a single `nat`. The second (dual) type expresses that a single `nat` should be sent on it. Communication can occur because one endpoint is captured by the closure that is executed in a new thread.

When communication occurs on an endpoint, the protocol described by the session type progresses. This is reflected in our small language by the fact that communication primitives return endpoint references with an updated type. The following typing rules for send and recv make this precise.

$$\frac{\Gamma \vdash e : a \,?\, \beta}{\Gamma \vdash \mathsf{recv}\ e : a \times \beta}\ \text{T-Recv} \qquad \frac{\Gamma_1 \vdash e_1 : a \,!\, \beta \qquad \Gamma_1 \sqcup \Gamma_2 \simeq \Gamma \qquad \Gamma_2 \vdash e_2 : a}{\Gamma \vdash \mathsf{send}\ e_1\ e_2 : \beta}\ \text{T-Send}$$

That is, recv takes an endpoint reference with protocol a ? β and returns a pair of an a and an updated endpoint reference with protocol β. The send command takes an endpoint reference with protocol a ! β and returns one with protocol β. To ensure type safety, subsequent communication *must* use the updated channel reference. For example, reusing the channel endpoint `l` in figure 4.1 of type `nat ! end` after sending the number on it violates *session fidelity* and breaks type safety. Alternatively, discarding the channel endpoint before sending a value on it leaves the other thread waiting, thus causing the program to be stuck. Hence, to attain type safety, one must ensure that channel references are used linearly—i.e., exactly once. Linearity is visible in the separation of the lexical context Γ into two disjoint parts Γ_1 and Γ_2 in the rule T-send (written $\Gamma_1 \sqcup \Gamma_2 \simeq \Gamma$), and is further enforced by limiting the shape of the context in the rules for literals and variables:[3]

Session fidelity: communication occurs according to the protocol described by the channel's session type

[3] Vasconcelos 2012. "Fundamentals of session types"

$$\frac{}{\varnothing \vdash n : \mathsf{nat}}\ \text{T-Nat} \qquad \frac{}{(x : a) \vdash x : a}\ \text{T-Var}$$

A CHANNEL ENDPOINT is an example of a reference that admits *strong update*. It is well-known that strong update is incompatible with sharing, and consequently we find linear type systems in other languages with strong update. We will use a linearly typed lambda calculus with linear references (LTLC_{ref}) as a case study, to demonstrate the negative impact of a linear type system and of linear references on the clarity of an intrinsically-typed interpreter.[4]

Strong update: Updates to a reference that may change the type of the reference, or delete the underlying cell.

[4] Although the linear references of LTLC_{ref} are too strict for practical purposes, it is useful for demonstrating the problem and our solution in a simple setting.

4.2 *Co-de-Bruijn Syntax for LTLC_{ref}*

We formalize the typed syntax of LTLC_{ref} as an inductive type family Exp in Agda. The formalization of the typed syntax is heavily informed by the typing rules. A crucial difference with the extrinsic typing presented above, is that we use a nameless representation of variable binding, as is the custom in mechanized representations of programming languages:

Agdaism: The underscores in a name defined in Agda denote where the arguments go. For example, the type for linear functions _⊸_ is used as $a \multimap b$.

```
data Ty : Set where
  unit  : Ty
  ref   : Ty → Ty
  prod  : Ty → Ty → Ty
  _⊸_   : Ty → Ty → Ty
```

In nameless representations of syntax, type contexts are represented by lists of types:

```
Ctx = List Ty
```

In the typing rules, the notion of disjoint context separation[5] $\Gamma_1 \sqcup \Gamma_2 \simeq \Gamma$ plays an important role. We can define context separation as an inductive ternary *proof-relevant relation*:

[5] Vasconcelos 2012. "Fundamentals of session types"

Proof-relevant relation: A relation in Set where different witnesses have different meaning.

```
data _⊔_≃_ : Ctx → Ctx → Ctx → Set where
  nil   : [] ⊔ [] ≃ []
  consˡ : Γ₁ ⊔ Γ₂ ≃ Γ → (t :: Γ₁) ⊔ Γ₂     ≃ (t :: Γ)
  consʳ : Γ₁ ⊔ Γ₂ ≃ Γ → Γ₁     ⊔ (t :: Γ₂) ≃ (t :: Γ)
```

The definition of the type family Exp then combines the grammar of the language with its typing rules, as usual:[6,7]

```
data Exp : Ty → Ctx → Set where
  unit  :                                          Exp unit []
  var   :                                          Exp t [ t ]
  lam   : ∀ t → Exp u (t :: Γ)                   → Exp (t ⊸ u) Γ
  app   : Exp (t ⊸ u) Γ₁ → (Γ₁ ⊔ Γ₂ ≃ Γ) → Exp t Γ₂ → Exp u Γ
  pair  : Exp t Γ₁ → (Γ₁ ⊔ Γ₂ ≃ Γ) → Exp u Γ₂     → Exp (prod t u) Γ
  ref   : Exp t Γ                                  → Exp (ref t) Γ
  swap  : Exp (ref t) Γ₁ → (Γ₁ ⊔ Γ₂ ≃ Γ) → Exp u Γ₂ → Exp (prod t (ref u)) Γ
  del   : Exp (ref unit) Γ                         → Exp unit Γ
```

[6] Note that the elimination of unit and pairs are omitted for brevity.

Compare unit, var and app with the typing rules of the session-typed language. The type of the constructor unit witnesses that the type context is empty. The representation of variables var is nothing more than the observation that the context is a singleton. The representation of binders that naturally obtain, is the *co-de-Bruijn* representation[8,9]. Whereas in a de-Bruijn representation, the choice between variables in scope is delayed until the leaves of the syntax tree, in a co-de-Bruijn representation, the choice is made at the earliest opportunity. That is, variables are only kept in the context of a subtree if they are used there.

[7] The operations swap and del are not monotone (chapter 3).

[8] McBride 2018. "Everybody's got to be somewhere"

[9] Abel et al. 2011. "A Lambda Term Representation Inspired by Linear Ordered Logic"

Because variables are nameless, all information about names is captured in context separation witnesses $\Gamma_1 \sqcup \Gamma_2 \simeq \Gamma$. To see why this relation *must* be proof-relevant, consider the possible inhabitants of the hole below. There are two, and each choice encodes different variable usage.[10]

```
tm : Exp (unit ⊸ unit ⊸ prod unit unit) []
tm = lam unit (lam unit (pair var σ var))
  where
    σ : [ unit ] ⊔ [ unit ] ≃ (unit :: unit :: [])
    σ = {!!}
```

[10] The two choices are:

σ_1 = cons^l (cons^r nil)
σ_2 = cons^r (cons^l nil)

corresponding to $\lambda x \to \lambda y \to (y, x)$ and $\lambda x \to \lambda y \to (x, y)$ respectively.

At run time we have a *store* of values, typed by a store type StoreTy. The cells of the store can be updated in ways that change the store type. Indeed, they can even be deleted! These strong updates are enabled by the key invariant of linear languages: the absence of sharing of lexical variables implies absence of sharing of references.

To intrinsically witness the absence of sharing in the runtime, we follow the same strategy for the specification of typed references as for lexical variables—i.e., we represent them using a co-de-Bruijn encoding:[11]

```
StoreTy = List Ty – Φ, Φ₁, Φ₂, etc.

data Val : Ty → StoreTy → Set where
  unit :                                          Val unit []
  ref  :                                          Val (ref t) [ t ]
  pair : Val t Φ₁ → (Φ₁ ⊔ Φ₂ ≃ Φ) → Val u Φ₂ → Val (prod t u) Φ
  clos : Exp t (u :: Γ) → Env Γ Φ                 → Val (t ⊸ u) Φ
```

[11] As in previous chapters we could define values using a function ⟦_⟧ by induction on the types, but we define it here as an indexed data-type Val, so that we can *implicitly* quantify existentially over store types (Φ_1 and Φ_2) and closure contexts Γ without defining separate records.

The StoreTy is to references what Ctx is to variables: it represents the consumption of the runtime context (i.e., the store type) by a value. The disjoint separation of store types Φ_1 and Φ_2 in the constructor for pairs witnesses that references are not shared between the two projections of the pair. We represent function values syntactically, as in section 3.6. The Env Γ Φ in the constructor clos is the *environment* captured by the closure value for the type context Γ. The definition of a typed environment is essentially the same as before, except that environments must internally maintain separation between values to witness linear use of the store:

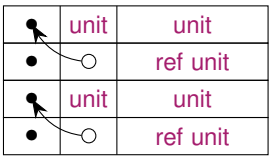
```
data Env : Ctx → StoreTy → Set where
  nil  : Env [] []
  cons : Val t Φ₁ → (Φ₁ ⊔ Φ₂ ≃ Φ) → Env Γ Φ₂ → Env (t :: Γ) Φ
```

Typed stores are structurally exactly like environments:

```
Store : StoreTy → StoreTy → Set
Store Φ₁ Φ₂ = Env Φ₁ Φ₂
```

where Φ_1 is the typing of the entire store, whereas Φ_2 is the store type that represents which part of the store is consumed by the store itself. A store of size 4 that stores two unit values and two pointers to those unit values is depicted in figure 4.2.

```
store : Store (unit :: ref unit :: unit :: ref unit :: []) (unit :: unit :: [])
store = cons unit (cons^r (cons^r nil))
      ( cons ref  (cons^l (cons^r nil))
      ( cons unit (cons^r nil)
      ( cons ref  (cons^l nil)
        nil
      )))
```



▲

Figure 4.2: An example encoded linear store and its graphical depiction. The first column represents the nameless locations, the second column contains the values of the cells, and the third column their types. Reference values are drawn as arrows.

The first index of this *store* types the four consecutive cells. The second index gives the typing of the part of the store that *is consumed* by the store itself—i.e., the cells that are referenced by other cells on the store.

An astute reader may remark that the interesting bit of information appears to be the *difference* between the two indices, being the *leftover* store that is available to clients of the store. We will encounter this 'difference' in the signatures of memory safe interpreters in the next and subsequent sections.

4.3 *Linear Interpreters*

By typing the source language, the target languages, and the semantic components, we can specify the type of an intrinsically-typed interpreter:

```
eval : Exp t Γ →
       Env Γ Φ₁ → (Φ₁ ⊔ Φ₂ ≃ Ψ) → Store Ψ Φ₂ →
       ∃⟨ (λ Ψ₂ Φ₃ Φ₄ → Store Ψ₂ Φ₃ × (Φ₃ ⊔ Φ₄ ≃ Ψ₂) × Val t Φ₄) ⟩
```

This type expresses: (1) type preservation: the value type t matches the expression type, (2) disjoint store consumption[12], and (3) absence of dangling references, because the total consumption equals the complete store. The existential quantification on the right-hand side is due to the fact that evaluation may add, remove or change cells in the store in statically unknown ways.

Agdaism: Recall that the syntax $\exists\langle\ P\ \rangle$ is an Agda idiom for existential quantification over all arguments of the parameterized type P. (See appendix A.)

[12] The input store is typed by Ψ, which is separated into the store consumption of the environment Φ_1 and the store consumption of the store itself Φ_2. The result store is typed by Ψ_2, which separates into the consumption Φ_3 of the result store, and the consumption Φ_4 of the result value.

Unfortunately, while correct as a top-level specification, the type of eval is not strong enough to evaluate effectful expressions inside a store that contains more than what the expression itself refers to. This occurs for example when we attempt to implement function application app $f\ e$: we want to evaluate the function expression f in the part of the environment that belongs to it. The remainder of the environment (for e) becomes a *frame* of the computation that is not passed to eval, but must remain separated from the values that the recursive call does manipulate. Consequently, the consumption of the environment Φ_1 and the store Φ_2 do not necessarily add up to the complete store type Ψ. We can generalize the type to additionally include a disjoint frame fr:

Figure 4.3: Direct formulation of the type of a type-safe linear interpreter and an example input and output store that fits the signature.

▼

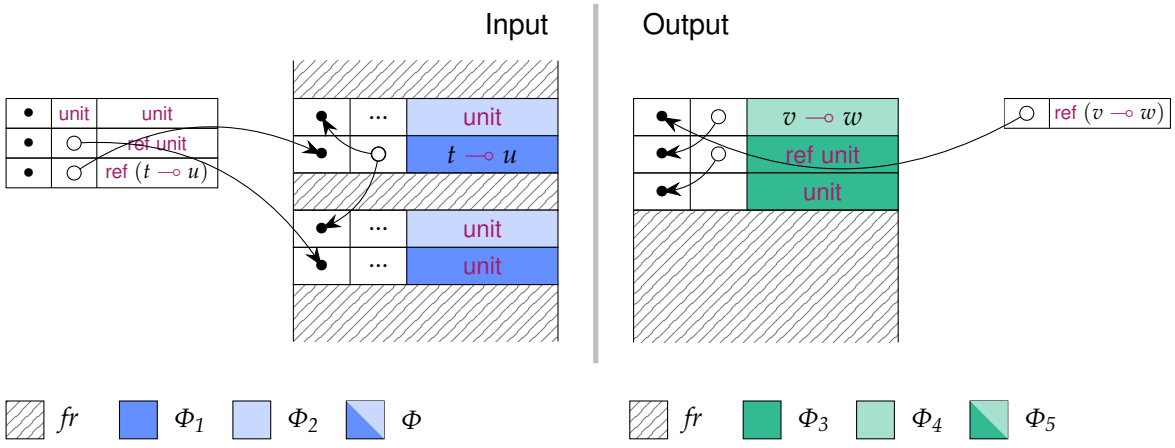
$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Exp}\ t\ \Gamma \to \\
&\quad \mathsf{Env}\ \Gamma\ \Phi_1 \to (\Phi_1 \sqcup \Phi_2 \simeq \Phi) \to \mathsf{Store}\ \Psi\ \Phi_2 \to \\
&\quad (\Phi \sqcup fr \simeq \Psi) \to \\
&\quad \exists \langle\ (\lambda\ \Psi_2\ \Phi_3\ \Phi_4\ \Phi_5 \to \mathsf{Store}\ \Psi_2\ \Phi_3 \times (\Phi_3 \sqcup \Phi_4 \simeq \Phi_5) \times \mathsf{Val}\ t\ \Phi_4 \times (\Phi_5 \sqcup fr \simeq \Psi_2))\ \rangle
\end{aligned}
$$

The key point is that the frame fr is *preserved* by the computation, such that pointers in the frame are not invalidated by eval. The frame preservation is observed by the separation $\Phi_5 \sqcup fr \simeq \Psi_2$).

An example input environment and store, together with an output store and value pair are displayed in figure 4.4. The fragmented consumption of the store typing, as well as the frame, are marked with color. Some cells in the input store are consumed by values in the environment. All remaining cells are held by values in the store itself, with the exception of the frame fr.

Figure 4.4: An example input and output of eval. The legend relates the store consumption to the signature in figure 4.3. Only the frame needs to be preserved.

▼



Unlike with monotone state, the output store need not at all contain the same cells as the input store (e.g., if the the input expression deletes a reference). Only the frame needs to be preserved by evaluation. Cells of the frame can be "moved" within the store data-structure, because the cells are not referenced by name.

The separation of the store type in the interpreter signature obscures what the interpreter does computationally. Implementations of this type are not any better: even if we look past pattern matching on an 8-tuple, the separation witnesses of the left- and right-hand side are not merely being passed around. For recursive calls to eval the separation witnesses have to be reassociated. Recursive evaluation may update the store, resulting in more separation witnesses and more proof obligations. The overhead of these proof terms obscures the computational content of the interpreter and makes writing them a tedious exercise. This manipulation of separation proofs in intrinsically-typed semantics of linear languages has previously been identified as a key issue of the approach[13].

[13] Thiemann 2019. "Intrinsically-typed mechanized semantics for session types"

The presence of an excessive amount of proof work in the interpreter comprises a large gap in clarity between Augustsson's interpreter for STLC (figure 2.2) or the interpreter for monotone state (figure 3.17), and an interpreter for a linearly-typed language. This chapter bridges that gap. The key idea is to use separation logic to build monadic abstractions with which we hide the explicit separation of the resource from types and implementations. The strong guarantees of these monadic operations can be made precise in their types without drowning the computational content in specifications of separation.

4.4 *Proof Relevant Separation Logic*

In section 4.3 we saw that explicitly writing out all typing contexts and the separation between them quickly becomes a tedious exercise, which obscures the clarity of the specification and implementation. In this section we construct a shallow embedding of a *separation logic* to hide the manipulation of linear contexts. We then show that separation logic can be used to concisely implement an intrinsically-typed interpreter for LTLC—the linearly-typed lambda calculus (section 4.5). In section 4.6 we will extend this interpreter to the state operations of LTLC_{ref}, and in section 4.8 we will extend this to the

operations for concurrency and communication of linearly typed lambda calculus with session primitives (LTLC_{ses}). Importantly, we are able to use the *same embedded logic* to hide the runtime resources of LTLC_{ref} and LTLC_{ref}, despite the fact that the invariant of linear references is more complicated than that of linear lambda-binding.

A WELL-KNOWN MODEL of separation logic can be given in terms of *predicates over a resource*[14] $\mathcal{R}$. We give a shallow embedding of such a model in Agda, but using *proof-relevant predicates*[15]:

```
Pred : Set → Set₁
Pred ℛ = ℛ → Set
```

We use proof-relevant predicates because we will cast intrinsically-typed syntax as such predicates:

```
data Exp : Ty → Pred Ctx
data Val  : Ty → Pred StoreTy
```

That is, the resource that we abstract over is a list of types. The list represents either the type context (for source syntax) or the store typing (for runtime objects).

RESOURCES need to have sufficient structure so that we can define the logical connectives of separation logic. The structure of the resource must also be lawful, so that the connectives of the logic also obey the usual laws of separation logic. We axiomatize resources as *proof relevant separation algebras* (PRSA). A PRSA is a quadruple $(\mathcal{R}, \_\bullet\_\simeq\_, \_\approx\_, \epsilon)$—being a carrier set, a separation or composition relation, an equivalence relation, and a unit, respectively. The quadruple must obey the laws of a partial, commutative, *proof-relevant* monoid. The laws of a PRSA are given with respect to the given (proof-relevant) equivalence relation $\_\approx\_$ on the carrier $\mathcal{R}$:

```
•-id^l    : (ϵ • a ≃ a)
•-id^-l   : (ϵ • a ≃ b) → a ≈ b
•-comm    : (a • b ≃ c) → (b • a ≃ c)
•-assoc   : (a • b ≃ ab) → (ab • c ≃ abc)
          → ∃⟨ (λ bc → (a • bc ≃ abc) × (b • c ≃ bc)) ⟩
```

In addition, the relation $\_\bullet\_\simeq\_$ must respect the equivalence relation $\_\approx\_$ in all three positions:

[14] O'Hearn et al. 1999. "The logic of bunched implications"

[15] This is in contrast to models of modern separation logics such as Iris, which use predicates in the universe of propositions (Jung et al. 2018b).

Agdaism: This is valid Agda *because* we use a shallow embedding. The Agda type checker can unfold Pred, such that these become valid data declarations.

```
•-respects-≈ˡ : a ≈ a′ → a • b ≃ c → a′ • b ≃ c
•-respects-≈ʳ : a ≈ b′ → a • b ≃ c → a • b′ ≃ c
•-respects-≈  : c ≈ c′ → a • b ≃ c → a • b  ≃ c′
```

We can define the PRSA of linear typing contexts, using list interleavings `_⨄_≃_` as the ternary composition relation, the empty list as a unit, and propositional equality as the equivalence relation. The laws are straightforward to prove in Agda.

We define the logical connectives of separation logic on predicates over elements of the resource $\mathcal{R}$.[16] In this order, we define the *separating conjunction* $*$, its unit Emp, and *magic wand* (or separating implication) $-\!\!*$:

```
record _*_ (P Q : Pred R) (r : R) : Set where
  constructor _•⟨_⟩_
  field
    {r₁ r₂} : R
    px      : P r₁
    split   : r₁ • r₂ ≃ r
    qx      : Q r₂

record _-*_ (P Q : Pred R) (r₁ : R) : Set where
field
  _⟨_⟩_ : ∀ {r₂ r} → r₁ • r₂ ≃ r → P r₂ → Q r
```

The separating conjunction $*$ concisely expresses that two predicates are conjoined disjointly. The $*$ associates to the left, and binds more tightly than its adjoint, the magic wand $-\!\!*$.

We encourage the reader to just think of wands as functions with an extra parameter that separates their argument from the 'surrounding' resources. To accommodate that intuition we make use of Agda's copatterns (see appendix A) to have all arguments of a wand on the left-hand side of a definition. For example:

```
mkTuple : ∀[ Q ⇒ P -* (Q * P) ]
mkTuple q ⟨ σ ⟩ p = q •⟨ σ ⟩ p
```

The context for the body of this definition is depicted in the margin. We should read this as a definition with three explicit arguments q, σ, and p (the resources r_1, r_2 and r are implicitly quantified). The arguments σ and p of this definition are the explicit arguments of the field `_⟨_⟩_` of the wand record. The witness σ separates p and q.

[16] In this thesis, we instantiate the logic for various concrete PRSAs. Whenever the PRSA is unambiguously determined, we will use the constructions from this section in an overloaded fashion.

Agdaism: The definition of $*$ uses a record to hide the existential quantification over the resources r_1 and r_2. The ternary constructor of the record has the left projection, the separation witness, and the right projection as arguments.

Agdaism: The definition of $-\!\!*$ uses a record to accommodate type inference.

```
r₁, r₂, r : R
q         : Q r₁
σ         : r₁ • r₂ ≃ r
p         : P r₂
```

Copatterns can be nested, so that this extends to nested wands as follows:

```
mkTriple : ∀[ R ⇒ Q —∗ P —∗ ((R ∗ Q) ∗ P) ]
mkTriple r ⟨ σ1 ⟩ q ⟨ σ2 ⟩ p = (r •⟨ σ1 ⟩ q) •⟨ σ2 ⟩ p
```

We define two predicates Own r and Emp that enable expressing *ownership* of a given resource r or the unit element ϵ, respectively:

```
Own : R → Pred R
Own r1 = λ r2 → r1 ≡ r2

Emp : Pred R
Emp = Own ε
```

Agdaism: Because Own and Emp are defined in terms of propositional equality ≡, inhabitants of these predicates are constructed by refl.

Besides the separation logic connectives, we also make use of the ordinary (pointwise) connectives on predicates. In particular, we use the pointwise arrow $P \Rightarrow Q$, the universal closure ∀[P] of a predicate, and the empty closure ϵ[P] of a predicate:

Figure 4.5: Pointwise arrows, universal-, and empty closure.

▼

```
_⇒_  : (P Q : Pred R) → Pred R
P ⇒ Q = λ r → P r → Q r

∀[_]  : Pred R → Set
∀[ P ] = ∀ {r} → P r

ε[_]  : Pred R → Set
ε[ P ] = P ε
```

The ∀ and ϵ closures are two ways to embed propositions from the logic in Agda. The fact that we use both has to do with the fact that a *pure* wand—i.e., one that does not own any resources—and a ∀-quantified pointwise arrow are equivalent:

```
wandit : ∀[ P ⇒ Q ] → ε[ P —∗ Q ]
unwand : ε[ P —∗ Q ] → ∀[ P ⇒ Q ]
```

In practice, we prefer ∀-quantified functions because they can be constructed as normal Agda functions and have one argument fewer. Similarly, there is an equivalence between the pointwise arrow that takes an argument Emp and the empty closure of a predicate. In practice we again prefer the latter, because it does not have an argument:

```
force : ∀[ Emp ⇒ P ] → ε[ P ]
abs   : ε[ P ]       → ∀[ Emp ⇒ P ]
```

Using the ∀-closure, we can spell out the usual logical laws[17],

[17] Reynolds 2002. "Separation logic: A logic for shared mutable data structures"

such as the adjunctive relationship between separating conjunction and the magic wand, and the unit laws of the separating conjunction:

$$\begin{aligned}
&\mathsf{uncurry} : \forall[\ (P * Q) \Rightarrow R\ ] \to \forall[\ P \Rightarrow (Q \mathrel{-\!\!*} R)\ ] \\
&\mathsf{curry} : \forall[\ P \Rightarrow (Q \mathrel{-\!\!*} R)\ ] \to \forall[\ (P * Q) \Rightarrow R\ ] \\
&*\text{-}\mathsf{id}^{l} : \forall[\ P \Rightarrow \mathsf{Emp} * P\ ] \\
&*\text{-}\mathsf{id}^{r} : \forall[\ P \Rightarrow P * \mathsf{Emp}\ ] \\
&*\text{-}\mathsf{id}^{-r} : \forall[\ P * \mathsf{Emp} \Rightarrow P\ ] \\
&*\text{-}\mathsf{id}^{-l} : \forall[\ \mathsf{Emp} * P \Rightarrow P\ ]
\end{aligned}$$

It is notoriously difficult to get some intuition for the magic wand connective. Cast into this programming setting, the resources that play a part in its definition can be understood as follows. A value of type $(P \mathrel{-\!\!*} Q)\ \Phi_1$ is a *function closure* capturing the resources Φ_1. Applying the closure requires an argument $px : P\ \Phi_2$ where the resource Φ_2 is disjoint from Φ_1. The application will yield a value $Q\ \Phi$ such that Φ is the sum of the resources Φ_1 and Φ_2, so that *no resources are lost*. The last point makes it a fitting tool for the specification of the invariants of linear languages.

We previously mentioned that typing contexts form a PRSA using list interleavings $\_ \sqcup \_ \simeq \_$ as the ternary composition relation. This is in stark contrast with previous notions of separation algebras, where lists have no natural instances. For example, separation algebras (SAs)[18] are a triple $(\mathcal{R}, J, \epsilon)$ that is associative, commutative and has ϵ as an identity of the ternary relation J, but is also cancellative and functional:

$$\begin{aligned}
&\bullet\text{-}\mathsf{cancel} : \mathsf{J}\ a_1\ b\ c \to \mathsf{J}\ a_2\ b\ d \to a_1 \equiv a_2 \\
&\bullet\text{-}\mathsf{func} : \mathsf{J}\ a\ b\ c_1 \to \mathsf{J}\ a\ b\ c_2 \to c_1 \equiv c_2
\end{aligned}$$

Other prior work instead uses an axiomatization based on a triple $(\mathcal{R}, \bullet, \epsilon)$ where the monoid operation $\_ \bullet \_$ is taken to be a partial function $\mathcal{R} \times \mathcal{R} \rightharpoonup \mathcal{R}$.[19,20] Both these notions of separation algebras prohibit instances that use arbitrary list interleavings as the composition/separation relation. When instances of these algebras for lists are defined, the lists are treated essentially as maps whose keys are the indices of the list. Positions in the list are thus relevant, and a "hole" at a position in the list is explicitly represented.[21]

All examples of PRSAs that we use in this thesis exploit proof

[18] Dockins et al. 2009. "A fresh look at separation algebras and share accounting"

[19] Calcagno et al. 2007. "Local action and abstract separation logic"

[20] Jung et al. 2018b. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic"

[21] Paykin et al. 2017. "The linearity Monad"

relevance and cannot be represented using one of the prior axiomatizations.

4.5 *Typing Linear Syntax and Functions using SL*

We can use the shallow embedding of our proof-relevant separation logic as a logical framework and give a concise intrinsically-typed syntax of the linearly typed lambda calculus (LTLC):[22]

```
data Exp : Ty → Pred Ctx where
  var  : ∀[ Own [ a ]              ⇒ Exp a          ]
  lam  : ∀[ ([ a ] ⊢ Exp b)        ⇒ Exp (a —∘ b)   ]
  app  : ∀[ Exp (a —∘ b) ∗ Exp a ⇒ Exp b          ]
  pair : ∀[ Exp a ∗ Exp b          ⇒ Exp (prod a b) ]
```

Here, variables are introduced into the body of the lambda by the following predicate transformer:

```
_⊢_ : Ctx → Pred Ctx → Pred Ctx
Γ ⊢ P = λ Δ → P (Γ ++ Δ)
```

The values of LTLC are pure and do not consume any runtime resource (like a store). We will nonetheless define values (and consequently also environments) as predicates over an abstract resource $\mathcal{R}$. This prepares us for adding references to the language, which we will do in section 4.6. Because linear references can also be encoded using the co-de-Bruijn representation, we use the exact same approach to specify the run time objects:

```
data Val : Ty → Pred ℛ where
  clos : Exp b (a :: Γ) → ∀[ Env Γ ⇒ Val (a —∘ b) ]
  pair : ∀[ Val a ∗ Val b ⇒ Val (prod a b) ]
```

Environments are naturally represented as indexed lists (indexed Kleene star) in this universe:[23]

```
data IStar (P : A → Pred ℛ) : List A → Pred ℛ where
  nil  : ϵ[ IStar P [] ]
  cons : ∀[ P a ∗ IStar P as ⇒ IStar P (a :: as) ]

Env : List Ty → Pred ℛ
Env = IStar Val
```

[22] The used technique of specifying typed syntax using type formers of context predicates was previously used for de Bruijn syntaxes by Allais et al. (2017). McBride (2018) presents a syntax for the co-de-Bruijn encoding of simply typed lambda calculus (STLC) that is virtually identical in appearance to our presentation of LTLC. Both directly inspired our contributions. See related work (section 4.10) for a discussion.

[23] Note how we use the ϵ-closure for the constructor nil to simplify the use of the data-structure. An equivalent but slightly more verbose type is:

```
nil : ∀[ Emp ⇒ IStar P [] ]
```

See also the discussion of figure 4.5.

This is the first of a number of familiar data structures and computational structures that we will transfer to our embedded separation logic. We will see that the linearity invariant of the runtime is nicely hidden by the logic, so that we obtain type signatures for interpreters and operations of LTLC that mirror their non-linear counterparts for STLC. The simplest example of this is how we can use a parameterized state monad to implement the effects of reading from (and pushing values into) the environment.

WE SPECIFY THE SEMANTICS of environment operations by means of a resource-aware parameterized[24] *predicate transformer*.[25]

```
IState : Ctx → Ctx → Pred R → Pred R
IState Γ1 Γ2 P = Env Γ1 —∗ Env Γ2 ∗ P
```

The parameters Γ_1 and Γ_2 of IState denote the shape of the environment before and after running the computation, respectively. The magic wand denotes that the environment that the reader computation expects, must be separated from any value that the computation closes over, while the separating conjunction means that the returned environment and value in P are separated.

Using IState we can type the interpreter for LTLC expressions analogous to a monadic interpreter for STLC:

```
eval : Exp a Γ → ϵ[ IState Γ [] (Val a) ]
```

The indices Γ and [] of IState denote that the computation consumes an environment of shape Γ entirely. The fact that the type is embedded with the ϵ-closure denotes that it does not depend on any resources outside of the reader computation.

THE EFFECTFUL OPERATIONS of LTLC are implemented by IState-operations. Modulo some getting used to, primitive operations are implemented much like normal Agda functions. As an example, we walk through the interactive development of the primitive get, starting with the declaration:

```
get : ϵ[ IState Γ [] (Env Γ) ]
get = {!!}
```

[24] Atkey 2009. "Parameterised notions of computation"

[25] We use a *state* monad, rather than a *reader* monad. The reason for this is simply that the most natural way to model linear use of the environment, is to *remove* a value from the environment when it is read. State computations return the part of the environment that is not read, so that nothing is discarded.

When we let Agda's interactive mode case split on the result type it expands the left-hand side, producing the following copattern:

```
get ⟨ σ ⟩ env = {!!}
```

The context and the type of the hole (i.e., the *goal*) is depicted in the margin. We can now let Agda refine the hole further and fill in the remaining holes based on the types:

```
get ⟨ σ ⟩ env = nil •⟨ σ ⟩ env
```

Agda thus accommodates the use of this shallowly embedded logic to some extent, so that the specification and implementation of operations in the logic is similar to that of normal Agda functions. Proofs that resources are preserved are carried out by relying heavily on (dependent) pattern matching. We will see further examples that also make use of the axioms of PRSAs.

$$\begin{array}{rcl} \Phi, \Phi_p & : & \mathcal{R} \\ \sigma & : & \epsilon \bullet \Phi_p \simeq \Phi \\ env & : & \mathsf{Env}\ \Gamma\ \Phi_p \\ \hline goal & : & (\mathsf{Env}\ [] * \mathsf{Env}\ \Gamma)\ \Phi \end{array}$$

Agdaism: The copattern (appendix A) introduces the wand record type by "projecting" the single field _⟨_⟩_ on the left-hand side.

OTHER PRIMITIVE OPERATIONS can be implemented similarly. By indexing reader computations with a pre and post environment, we can specify computations that do not consume the entire environment. Consequently, we can also specify operations that extend the environment:

```
prepend : ∀[ Env Γ₁ ⇒ IState Γ₂ (Γ₁ ++ Γ₂) Emp ]
append  : ∀[ Env Γ₁ ⇒ IState Γ₂ (Γ₂ ++ Γ₁) Emp ]
```

Furthermore, we can use the fact that context separation implies environment separation to frame reader computations inside larger environments:

```
frame : Γ₁ ⊔ Γ₃ ≃ Γ₂ → ∀[ IState Γ₁ [] P ⇒ IState Γ₂ Γ₃ P ]
```

That is, we split the environment typed by Γ_2 into two disjoint parts of type Env Γ_1 and Env Γ_3, and run the state computation on the former, returning the latter.[26]

In an untyped setting we would use the fact that IState is a monad in Set to compose these operations into larger computations. However, it is a priori unclear in what sense IState is a monad. A first attempt is to implement the interface of a parameterized monad:

```
return :  ∀[ P ⇒ IState Γ Γ P ]
bind   :  ∀[ P ⇒ IState Γ₂ Γ₃ Q ]
       → ∀[ IState Γ₁ Γ₂ P ⇒ IState Γ₁ Γ₃ Q ]
```

[26] One way to think about this is as exploiting a morphism between the PRSA for context separation and the PRSA for $\mathcal{R}$, as in the work of Farka et al. (2021).

Unfortunately, this bind is not strong enough by itself to implement an interpreter for LTLC. This can be seen more easily if we recall the equivalence between pointwise-lifted functions and magic wands, and rewrite the bind as:

$$\begin{aligned}&\mathsf{bind} : \ \epsilon[\ P \multimap^* \mathsf{IState}\ \Gamma_2\ \Gamma_3\ Q\ ] \\ &\quad \to \epsilon[\ \mathsf{IState}\ \Gamma_1\ \Gamma_2\ P \multimap^* \mathsf{IState}\ \Gamma_1\ \Gamma_3\ Q\ ]\end{aligned}$$

That is: we can only bind functions that do not close over any resources. This is insufficient, for example, for interpreting binary expressions (e.g., function application), where bound continuations close over previously computed resourceful values. To remedy this, we can *internalize* the bind:

$$\begin{aligned}&\mathsf{bind} : \forall[\ (P \multimap^* \mathsf{IState}\ \Gamma_2\ \Gamma_3\ Q) \\ &\quad \Rightarrow (\mathsf{IState}\ \Gamma_1\ \Gamma_2\ P \multimap^* \mathsf{IState}\ \Gamma_1\ \Gamma_3\ Q)\ ]\end{aligned}$$

This *internal bind* is strong enough to implement the interpreter, but the use of magic wands also has a downside: the fact that a magic wand takes a separation witness as an additional argument, means that every step in the interpreter will receive a proof term that needs to be passed around. As a side effect, we can also not use Agda's builtin do-notation, which expects a bind with the usual arity. Fortunately, we can again rely on the same programming trick that we also used to program with monads in the category of monotone predicates in chapter 3. We use the fact that a monad with an internal bind is equivalent to a strong monad[27,28]—i.e., we use the "external" bind and *tensorial strength* over the separating conjunction:

$$\mathsf{str} : \forall[\ Q * \mathsf{IState}\ \Gamma_1\ \Gamma_2\ P \Rightarrow \mathsf{IState}\ \Gamma_1\ \Gamma_2\ (Q * P)\ ]$$

We abbreviate $\mathsf{str}\ (q \bullet\langle\ \sigma\ \rangle\ m)$ as $q\ \&\langle\ \sigma\ \rangle\ p$, mirroring the notation from chapter 3 for tensorial strength in the category of monotone predicates. Using tensorial strength, we do not have to close the bound continuation over any outside resources. Instead, we pass them to the continuation *through* bind. The result is that despite the rich and complicated underlying structure of separation logic, we can program with these monads as regular monads in Agda.

[27] Kock 1972. "Strong functors and monoidal monads"

[28] Moggi 1991. "Notions of computation and monads"

The following excerpt of the LTLC interpreter shows how we combine reader operations to interpret linear function application:

```
eval : Exp a Γ → ϵ[ IState Γ [] (Val a) ]
eval (app (f •⟨ σ₁ ⟩ e)) = do
  clos b env     ← frame σ₁ (eval f)
  env •⟨ σ₂ ⟩ v ← env &⟨ •-id^r ⟩ eval e
  refl           ← append (cons (v •⟨ •-comm σ₂ ⟩ env))
  eval b
```

```
f   :  Exp (a ⊸ b) Γ₁
σ₁  :  Γ₁ ⊔ Γ₂ ≃ Γ
e   :  Exp a Γ₂
b   :  Exp b (a :: Γ₃)
env :  Env Γ₃ r₁
σ₂  :  r₁ • r₂ ≃ r
v   :  Val b r₂
```

Using frame, we first evaluate the function expression in a part of the environment prescribed by the context separation σ_1. By dependent pattern matching we obtain the corresponding closure clos b env. This leaves exactly the part of the environment that is required to evaluate the argument e. To get evidence σ_2 that the previously obtained closure environment is separated from the resulting value v, we use tensorial strength. This evidence is required to construct the environment for the body b. The indices of the linear IState monad ensure that we have constructed an environment that matches the context for the body.

With minimal proof overhead we have implemented a resourceful version of function application and proven it type safe. Moreover, any violation of any of the properties that we want to hold, is caught by Agda during the development of the interpreter. For example, appending v and env in the wrong order would be caught because the shape of the environment would not match the context of b. Or, if we were to forget v and put any other value in its place, Agda would not accept the definition, because some resources were lost.

In the process we have developed reusable abstractions for resourceful representations of data, and resource-aware computational structures. Just like the usual parameterized state monad, IState can be generalized to a (strong) monad transformer:

```
IStateT : (Pred ℛ → Pred ℛ) → Ctx → Ctx → Pred ℛ → Pred ℛ
IStateT M Γ₁ Γ₂ P = Env Γ₁ —∗ M (Env Γ₂ ∗ P)
```

We will use this transformer in the next section to extend the interpreter to the state operations of LTLC_{ref}.

4.6 *Intrinsically-Typed Linear References*

Memory safety: There are no dangling references

In section 4.3 we noted that the explicit signature of a definitional interpreter expresses not just type preservation and disjoint consumption of resources, but also *memory safety*. These last two properties were encoded together into a single observation that must hold on both sides of the evaluation signature: the total amount consumed, including the frame and the consumption of the store itself, must combine exactly to what the store provides.

Recall from section 4.2 that Store Ψ Φ is a store with cells typed by Ψ, and a combined consumption Φ. We will call Ψ the *supply* of the store, and Φ its *demand*. The runtime invariant for linearity is that the total demand of all *consumers* of the store add up to the supply. For example, looking at the explicit type of the semantic operation mkref—which implements the semantics of ref—we have three consumers: a frame, a value and the input store:

$$
\begin{aligned}
\mathsf{mkref} : \mathsf{Val}\ a\ \Phi_1 \to (\Phi_1 \sqcup \Phi_2 \simeq \Phi) \to \mathsf{Store}\ \Psi_1\ \Phi_2 \\
\to (\Phi \sqcup \mathit{fr} \simeq \Psi_1) \\
\to \exists \langle\ (\lambda\ \Psi_2\ \Phi_3\ \Phi_4\ \Phi_5 \to \mathsf{Store}\ \Psi_2\ \Phi_3 \\
\times\ \Phi_3 \sqcup \Phi_4 \simeq \Phi_5 \times \mathsf{Val}\ (\mathsf{ref}\ a)\ \Phi_4 \\
\times\ \Phi_5 \sqcup \mathit{fr} \simeq \Psi_2)\ \rangle
\end{aligned}
$$

The demand of the three consumers (fr, Φ_1, and Φ_2, respectively) add up to the total left-hand side supply Ψ_1.

The separation logic that we presented is by itself insufficient to express this notion of memory safety. In particular, while the $*$ can express the disjoint distribution of *demand* for store cells, it does *not* provide a means to equate this with the *supply* of the actual store. In this section we develop the abstractions to *balance* supply and demand.

We also want to hide all of the separation accounting. To that end, we need a PRSA that accounts for both supply and demand. Although we can construct a product PRSA, this will not suffice, because it will account for supply and demand in isolation of each other, and not enforce the top-level equation between them. Additionally, if we account for the supply and demand separately, the left- and right-hand side supply for an operation like mkref is not going to balance. The key to balancing the equation is not to look at supply and demand separately, but at their difference. The operation mkref is memory-safe, because it *increases* supply and demand *equally*

when it returns a pointer to the freshly allocated cell.

We will do this by using predicates not over store types StoreTy, but over a novel resource *transformer* Market. We define the Market PRSA transformer and show how it can be used for intrinsically-typed programming with linear state that supports strong update. In section 4.7, we then interpret the linear state operations of LTLC_{ref}.

THE PRSA Market $\mathcal{R}$ tracks the *net supply* of the PRSA $\mathcal{R}$. We first define the carrier of the algebra:

```
data Market (ℛ : Set) : Set where
  supply  : (s : ℛ) → Market ℛ
  demand : (d : ℛ) → Market ℛ
```

The algebra accounts for multiple consumers (each representing some demand) and *at most one* supplier (representing a supply) for $\mathcal{R}$. By restricting to a single supplier we can apply a simple accounting method operating in two modes, formalized as a separation relation $\_\bullet_m\_\simeq\_$ on Market $\mathcal{R}$ with three constructors:

```
data _•ₘ_≃_ : Market ℛ → Market ℛ → Market ℛ → Set where
```

If no supplier is present, then we are simply adding up demand:

```
  demands :  (d₁ • d₂ ≃ d)
             → (demand d₁) •ₘ (demand d₂) ≃ (demand d)
```

Or, if a supplier is present, we subtract the demand from it, tracking how much supply is left over:

```
  supplyₗ  : (s₂ • r ≃ s₁) → (supply s₁)  •ₘ (demand r) ≃ (supply s₂)
  supplyᵣ  : (r • s₂ ≃ s₁) → (demand r) •ₘ (supply s₁)  ≃ (supply s₂)
```

Importantly, there is no constructor that permits a supplier to be present on *both* sides. This ensures that if a single supplier is present, then every reference is bound in that supplier. The quadruple $(\text{Market } \mathcal{R}, \_\bullet_m\_\simeq\_, \_\approx_m\_, \text{demand } \epsilon)$ is a PRSA for every PRSA $(\mathcal{R}, \_\bullet\_\simeq\_, \_\approx\_, \epsilon)$, where $\_\approx_m\_$ just lifts the equivalence on $\mathcal{R}$ onto the constructors of Market $\mathcal{R}$.

The predicates Store, Val, and One can be lifted into Market StoreTy, provided that we clarify the role of these types as either suppliers or consumers. We define two modalities ($\bullet\_$ and $\circ\_$) for lifting suppliers and consumers, respectively:

```
data ● (P : Pred R) : Pred (Market R) where
  supplier : P s → ● P (supply s)

data ○ (P : Pred R) : Pred (Market R) where
  consumer : P d → ○ P (demand d)
```

Both ● and ○ associate more tightly than arrows and wands.

We now give a better definition of a well-formed store, incorporating the idea that we are keeping track of leftover supply:

```
record LeftOver (P : R → R → Set) (s₁ : R) : Set where
  constructor subtract
  field
    {s₂ d} : R
    px : P s₂ d
    sub : d • s₂ ≃ s₁
Store : Pred StoreTy
Store = LeftOver (IStar Val)
```

The type LeftOver takes a predicate indexed by both a supply s_2 and its internal demand d, and is itself indexed by the supply s_1 that is left over after subtracting that internal demand from the supply. The internal demand cannot exceed the supply for *positive* resources $\mathcal{R}$. This is the gist of the internal resource accounting of the store. The internal representation of stores (Star Val) are essentially the same as the stores of section 4.2 and as value environments Env. indexed list of values. We stress that the store type index of the Store then represents the *leftover supply* of the store that can be referenced by external clients of this linear state.

USING THESE TYPE FORMERS, we can now consisely express the type- and memory-safe signature of the operation mkref:

Compare this with the signature of mkref in the beginning of this section!

```
mkref : ∀[ ○ (Val a) ⇒ ● Store —∗ ○ (Own [ a ]) ∗ ● Store ]
```

The supply and demand equation will balance: the fact that the store is extended by a single new cell of type a, is offset by the returned pointer ○ (Own [a]).

By working in the Market-monoid, and by using the embedded separation logic connectives, we have rid the signature of a linear state operation of all the proof side-conditions so that it takes almost its usual shape. We now turn to defining a State predicate transformer, where we will see that it can hide the entire state invariant,

including the modalities. A client of this State monad will not have to be aware of the underlying use of the Market PRSA transformer to maintain memory safety. It will only be aware of the part of the invariant that it understands: ensuring that references are never shared.

We now define a State monad, taking special care to define it as an endofunctor in Pred $\mathcal{R}$. The straightforward definition does *not* accomplish this:

```
State : Pred R → Pred R → Pred (Market R)
State S P = ● S —∗ ○ P ∗ ● S
```

Which is not an endofunctor at all. We can easily define it as an endofunctor on Pred (Market $\mathcal{R}$), but in that case the user must always specify the consumer modality on the predicates in State, and must wrap and unwrap inputs and outputs of State operations.

We remedy this by making explicit that any wand that takes a *supplier* as an argument, can itself only be a *consumer*. That is, for any $f : (● P \multimap Q)\ \Phi_1$ it must be the case that Φ_1 is equal to demand Φ_2 for some Φ_2. This holds, because the resources that the wand uses must be separated from the resources used by the argument, which is fixed to be supply by the ●-modality. Without loss of generality, we can thus define the State monad as an endofunctor in Pred $\mathcal{R}$ as follows:

```
State : Pred R → Pred R → Pred R
State S P r = (● S —∗ ○ P ∗ ● S) (demand r)
```

This predicate transformer is a strong monad for any notion of state indexed by both supply and demand, and generalizes to a (strong) monad transformer StateT. The operations mkref, read, and update can be implemented if the state type is instantiated to Store. All operations are parametric in the type of values, and the notion of separation between them:

```
mkref  : ∀[ Val a                    ⇒ State Store (Own [ a ]) ]
read   : ∀[ Own [ a ]                ⇒ State Store (Val a) ]
write  : ∀[ Own [ a ] ⇒ Val b —∗ State Store (One b ∗ Val a) ]
update : ∀[ Own [ a ]
          ⇒ (Val a —∗ State Store (Val b))
          —∗ State Store (Own [ b ]) ]
```

Note that the signature of read tells us that the cell pointed to by the passed reference is destroyed, as the reference is not returned

from this operation. In contrast, the write operation keeps the cell, returning a pointer with its new type, and also the value that used to be in it. The update operation is an example of a higher-order operation, for which the use of a $-\!\!*$ is a necessity.

4.7 *Interpreting LTLC$_{ref}$*

To interpret LTLC$_{\text{ref}}$, we first extend the data types for expressions and values of LTLC with the state primitives:

```
data Exp : Ty → Pred Ctx where
  ref    : ϵ[ Exp a ⇒ Exp (ref a) ]
  swaps  : ∀[ Exp (ref a) ∗ Exp b ⇒ Exp (prod a (ref b)) ]
  del    : ϵ[ Exp (ref a) ⇒ Exp a ]

data Val : Ty → Pred StoreTy where
  pair : ∀[ Val a ∗ Val b ⇒ Val (prod a b) ]
  ref  : ∀[ Own [ a ]     ⇒ Val (ref a)    ]
```

We instantiate the linear State monad for stores over these values, and nest it inside the linear IState monad transformer for threading the linear environment. The state operations of LTLC$_{\text{ref}}$ are then implemented as follows:

```
eval : Exp a Γ → ϵ[ IStateT (State Store) Γ [] (Val a) ]
eval (ref e) = do
  v ← eval e
  r ← liftM (mkref v)
  return (ref r)

eval (swaps (e1 •⟨ σ ⟩ e2)) = do
  ref ra          ← frame σ (eval e1)
  ra •⟨ σ1 ⟩ vb ← ra &⟨ •-id^r ⟩ eval e2
  rb •⟨ σ2 ⟩ va ← liftM (write ra ⟨ σ1 ⟩ vb)
  return (pair (va •⟨ •-comm σ2 ⟩ ref rb))

eval (del e) = do
  ref r ← eval e
  liftM (read r)
```

The implementation of swaps first interprets the left- and right-hand side sub-expressions, from which we obtain a reference and a value. We also get a witness that these are separated by using tensorial

strength in the second step. This witness is needed in the subsequent evaluation step. Using liftM, we lift the write operation of the State monad into the reader transformer. From the write, we again obtain a reference and a value, separated according to σ_2. All that remains is to construct the pair value, which we return.

After context separation, we have now constructed a second proof-relevant separation algebra: Market $\mathcal{R}$, formalizing the accounting of supply and demand of some underlying resource $\mathcal{R}$. We have used this to construct a monad State with the familiar semantic operations on typed stores, transported to the linear setting. Remarkably, the complexities of the underlying accounting is hidden from the user entirely.

4.8 Intrinsically-Typed Sessions

We now turn to the session-typed language LTLC_{ses}, and its operations for spawning threads and conducting communication. We stage the interpretation into two layers. The first layer interprets the expression language into *command trees*[29], interleaving the communication and threading commands with thunked evaluation of the expression language. The second layer interprets these command trees, thus implementing the scheduling and communication semantics.

[29] Hancock et al. 2000. "Interactive programs in dependent type theory"

To implement these stages, we apply the abstractions that we developed so far. The syntax of the language, being an extended linearly-typed lambda calculus, again uses the list PRSA to deal with co-de-Bruijn variables. The monad for the expression language interpreter nests a novel free monad inside the reader transformer. For the command semantics, we reuse the Market PRSA and the state monad transformer StateT. We combine those with an Error monad to handle the partiality of receiving messages, and we use a new PRSA within Market to split channels into their endpoints.

The benefit of staging the interpretation, is that we clearly separate the runtime (i.e., the communication and concurrency model) from the language. The first and second stage are independent of each other and reusable. We will only implement round-robin scheduling and asynchronous communication, but one could swap this semantics of command trees for other schedulers and/or a synchronous communication model, without adjusting the expression language interpreter.

We embed the syntax of the concurrency and communication primitives in Agda, starting by extending the types of LTLC with a type for *channel endpoint references*, which we write endp α. We again use greek variables to denote session types, and we write $\alpha\ ^{-1}$ for the dual of a session type.

```
data STy : Set where
  end  : STy
  _?_  : Ty → STy → STy
  _!_  : Ty → STy → STy

data Ty : Set where
  endp : (α : STy) → Ty
```

```
_⁻¹ : STy → STy
(a ! β) ⁻¹ = a ? β ⁻¹
(a ? β) ⁻¹ = a ! β ⁻¹
end ⁻¹ = end
```

Session type constructors for sending and receiving are written in an infix style, for example $a\ !\ \beta$ for the protocol that sends an a and continues as β. Dual is involutive and has end as a neutral element:

```
dual-involutive : ∀ {α} → α ⁻¹ ⁻¹ ≡ α
dual-neutral    : ∀ {α} → end ≡ α ⁻¹ → α ≡ end
```

Agdaism: The fact that end $^{-1} \equiv$ end is true definitionally, so we formulate a more useful variation.

We then extend LTLC with the five primitive operations for spawning threads and conducting communication. The communication primitives all act on *channel endpoint references* and require the right protocol shape:

Figure 4.6: Primitives for multi-threading and session-typed communication.

▼

```
data Exp : Ty → Pred Ctx where
  fork  : ∀[ Exp (unit ⊸ unit) ⇒          Exp unit                          ]
  mkchan : ∀ α → ϵ[                      Exp (prod (endp α) (endp (α ⁻¹))) ]
  recv  : ∀[ Exp (endp (a ? β)) ⇒        Exp (prod (endp β) a)             ]
  send  : ∀[ Exp a ∗ Exp (endp (a ! β)) ⇒ Exp (endp β)                      ]
  close : ∀[ Exp (endp end) ⇒            Exp unit                          ]
```

We will implement a semantics for these primitive operations in section 4.9. The runtime will maintain a collection of channels, similar to how the runtime of LTLC_{ref} maintained a collection of cells. Unlike cells however, open channels have two handles which can be referenced independently: one handle for each endpoint. Hence, we model the session (or runtime) context SCtx as a list of *runtime types*[30], which can be either a single endpoint, or an entire channel consisting of two typed endpoints:

[30] Fowler et al. 2019. "Exceptional asynchronous session types: Session types without tiers"

```
data Runtype : Set where
  endpr : STy → Runtype
  chanr : STy → STy → Runtype

SCtx = List Runtype
```

Although conceptually channel endpoints have dual types, in practice, for buffered communication, this may not be the case.[31] We explain this in more detail in section 4.9. Mere interleavings of SCtx do not describe all the ways that session contexts can be split. In particular, we have to account for the separation of channels into their respective endpoints. We model the separation of individual channels using a ternary relation Ends:

```
data Ends : Runtype → Runtype → Runtype → Set where
  lr : Ends (endpr α) (endpr β) (chanr α β)
  rl : Ends (endpr β) (endpr α) (chanr α β)
```

We then define a PRSA on lists of a resource type—i.e., interleavings with an additional constructor divide for making ends meet:

[31] Fowler et al. 2019. "Exceptional asynchronous session types: Session types without tiers"

Figure 4.7: Generalized list separation. Disjoint list separation _⊔_≃_ can be recovered by choosing the element division relation _•_≃_ to be the empty relation.

▼

```
data _•ₗ_≃_ : (xs ys zs : List R) → Set where
  nil     :                  []      •ₗ  []       ≃ []
  consˡ   : xs •ₗ ys ≃ zs →  (z :: xs) •ₗ  ys       ≃ (z :: zs)
  consʳ   : xs •ₗ ys ≃ zs →  xs        •ₗ  (z :: ys) ≃ (z :: zs)
  divide : r₁ • r₂ ≃ r → (xs •ₗ ys ≃ zs) → (r₁ :: xs) •ₗ (r₂ :: ys) ≃ (r :: zs)
```

This quadruple (List $\mathcal{R}$, _•ₗ_≃_, _≈ₗ_, []) is indeed a PRSA whenever ($\mathcal{R}$, _•_≃_, _≈_) is a proof-relevant partial semigroup[32], where _≈ₗ_ extends _≈_ pointwise over lists. The type Ends is indeed such a proof-relevant partial semigroup.[33]

When we instantiate the above list PRSA with the partial semigroup Ends, then we get a relation that separates lists of Runtype elements in more ways than just interleaving them. For example, we can prove the following separations:

[32] We have not defined such triples formally here, but their definition can be recovered by dropping the unit and unit-laws from the definition of a PRSA. The Agda library does of course define it properly.

[33] It does not have a unit element and hence is not a PRSA.

```
ex₁ : (endpr α :: endpr β :: []) •ₗ (endpr γ :: []) ≃ (endpr α :: endpr γ :: endpr β :: [])
ex₁ = consˡ (consʳ (consˡ nil))

ex₂ : (endpr α :: endpr β :: []) •ₗ (endpr γ :: []) ≃ (chanr α γ :: endpr β :: [])
ex₂ = divide lr (consˡ nil)

ex₃ : (endpr α :: endpr β :: []) •ₗ (endpr γ :: []) ≃ (endpr α :: chanr γ β :: [])
ex₃ = consˡ (divide rl nil)
```

Example ex_1 is a mere disjoint separation of a lists of three endpoints. However, examples ex_2 and ex_3 each split a channel into its two endpoints and hand them to the left and right session contexts respectively.[34] Hence, the separation of session contexts is more involved than the separation of typing contexts and manually bookkeeping such environments is thus also more tedious.

[34] These session context separations coincide with the manual treatment of session contexts in the typing of the runtime in, for example, Fowler et al. (2019).

In the same way that values of LTLC_{ref} are predicates over a store type StoreTy, values and other runtime objects of the session-typed language are predicates over SCtx. As before, references (to endpoints) are typed in a co-de-Bruijn style:

```
data Val : Ty → Pred SCtx where
  cref : ∀[ One (endpr α) ⇒ Val (endp α) ]
```

Expressions are not interpreted to plain values, but to command trees[35,36] with values at the leaves. Outgoing commands may contain channel endpoint references, and must therefore be separated from their continuation. This yields the following strong monad Free P of command trees that will yield an instance of P:

[35] Hancock et al. 2000. "Interactive programs in dependent type theory"

[36] Swierstra et al. 2019. "A predicate transformer semantics for effects (functional pearl)"

```
data Cmd : Pred SCtx where
  fork    :                ∀[ Free (Val unit)                  ⇒ Cmd ]
  mkchan  : ∀ (β : STy) → ϵ[                                     Cmd ]
  send    : ∀ a β       → ∀[ Own [ endpr (a ! β) ] ∗ Val a ⇒ Cmd ]
  recv    : ∀ a β       → ∀[ Own [ endpr (a ? β) ]          ⇒ Cmd ]
  close   :                ∀[ Own [ endpr end ]              ⇒ Cmd ]

Resp : Cmd Φ → Pred SCtx
Resp (fork _)       = Emp
Resp (mkchan β)     = Own [ endpr β ] ∗ Own [ endpr (β ⁻¹) ]
Resp (send _ β _)   = Own [ endpr β ]
Resp (recv a β _)   = Val a ∗ Own [ endpr β ]
Resp (close _)      = Emp

data Free (P : Pred SCtx) : Pred SCtx where
  pure   : ∀[ P ⇒ Free P ]
  impure : ∀[ Σ[ c ∈ Cmd ]∗ (Resp c —∗ Free P) ⇒ Free P ]
```

Command trees Free P are trees whose nodes (impure) contain a command and whose branches are continuations for every possible response.[37] The set of possible commands is defined using the

[37] For readability we specialize Free here for a given type of commands and responses. In the library it is parameterized over those types.

predicate Cmd. Every primitive of threading and communication is represented by a command. Commands may contain resourceful values—e.g., a reference to a channel endpoint. The type of possible responses to a command c is Resp c. For example, the response to the mkchan command are two channel endpoint references with dual session types. Responses may again contain resourceful values. The specification makes use of a *dependent separating conjunction* for the pair of the command and the command-dependent type of the continuation:

Figure 4.8: Dependent separating conjunction.

▼

```
record Conj (P : Pred ℛ) (Q : ∀ {r} → P r → Pred ℛ) (r : ℛ) : Set where
  field
    {r₁ r₂} : ℛ
    px      : P r₁
    sep     : r₁ • r₂ ≃ r
    qx      : Q px r₂
```

We write Σ[$x \in P$]∗($Q\ x$) for Conj P ($\lambda\ x \to Q\ x$). The normal separating conjunction ∗ is defined in terms of the dependent version.

Commands and responses define the interface of the runtime. The type of the constructor impure exactly captures the exchange of resources that occurs between a thread and the runtime: when a command is issued, a thread gives away some resources via the command, and keeps the remainder enclosed in the continuation. The magic wand in the type of the continuation denotes that the thread may receive new resources via the response, separated from what it already owned. A command can be lifted to a computation in Free that will return the response:[38]

```
give : ∀ {Φ} → (c : Cmd Φ) → Free (Resp c) Φ
```

[38] We have not introduced syntax for dependent predicate implication, so we write the full signature of give here.

Because we defined the command fork to take a computation in Free rather than an expression Exp, we can define the interpretation of expressions and command trees entirely independently. The first stage is straightforward and only interprets the effect of reading the environment. By nesting Free inside the reader transformer, we can interpret all the effects of LTLC_{ses}.

We write out the evaluation of send and receive:

```
eval : Exp a Γ → ϵ[ IStateT Free Γ [] (Val a) ]
eval (recv e) = do
  cref ch1              ← eval e
  v •⟨ σ ⟩ ch2          ← liftM (give (recv _ _ ch1))
  return (pair (cref ch2 •⟨ •-comm σ ⟩ v))
eval (send (e1 •⟨ σ ⟩ e2)) = do
  v1                    ← frame σ (eval e1)
  v1 •⟨ σ ⟩ cref ch1 ← v1 &⟨ •-id^r ⟩ eval e2
  ch2                   ← liftM (give (send _ _ (ch1 •⟨ •-comm σ ⟩ v1)))
  return (cref ch2)
```

Again, the hard work of maintaining separation is hidden. Additionally, using the free monad construction, the concurrency is hidden. Finally, the fact that receiving a message on a channel is a blocking operation and may have to wait for the corresponding sent is completely opaque. The implementation of concurrency and communication is completely up to the second stage interpreter.[39]

[39] In this interpreter, threads only yield control when they send a command. More fine-grained concurrency can be achieved by adding a yield command and manually inserting it where desired, or incorporating it into the bind of the stage 1 interpreter.

This completes the first half of the semantics, which reduced the language to a tree of concurrency and communication primitives which have to be implemented by the runtime. The runtime semantics is the subject of the next section. There we show how to handle each of the runtime commands using the linear State monad from section 4.6 to implement the channel state.

4.9 *Interpreting Command Trees as Processes*

By interpreting the expression language to command trees, we have given an operational semantics for everything except the five primitive operations for concurrency and communication. These are the operations that operate on the runtime state of the language: the collection of channels and the threadpool. In this section we implement the runtime top-down, starting with a scheduler built on top of an abstract monadic interface. We work our way towards the implementation of the various communication primitives. We define the channel state consisting of linked buffers and finally implement the monadic interface using a monad transformer stack.

Threads are simply suspended computations—represented using the Free monad—that return values. We distinguish the main thread (which returns a value), from forked threads (which return unit).[40]

[40] Fowler et al. 2019. "Exceptional asynchronous session types: Session types without tiers"

```
data Thread : Pred SCtx where
  forked : ∀[ Free (Val unit) ⇒ Thread ]
  main   : ∀[ Free (Val a)    ⇒ Thread ]
```

To represent the thread pool, we use a datatype akin to lists or Kleene Star:

```
data Star (P : Pred A) : Pred A where
  nil  : ϵ[ Star P ]
  cons : ∀[ P ∗ Star P ⇒ Star P ]

Pool = Star Thread
```

The runtime can be understood as combining a scheduler—which selects a thread that can make a step—with a function that actually takes a step in a given thread.

We will interface with the scheduler a monadic operation dequeue which either returns a thread that is not yet done (inj_2 *thr* in subsequent code) or—if no such thread exists—returns an exception (inj_1 *e*). Threads can be added to the pool of the scheduler by an operation enqueue. A thread trying to make a step will also be allowed to throw an exception to signal that it cannot currently make a step—because it is blocked on a receive command while there is not yet a value in the receiving endpoint buffer—and should be delayed.

Assuming those primitive operations we can implement the following abstract scheduler:

```
run : ϵ[ M Emp ]
run = do
  thr? ← dequeue
  case thr? of λ where
    (inj₁ e) → return e
    (inj₂ thr) → do
      refl ← (do
        thr′ ← step thr
        enqueue thr′) orElse (enqueue thr)
      run
```

For clarity we elide the use of fuel to satisfy the termination checker, as well as the error handling for running out of fuel.

For round-robin scheduling, the monadic operations dequeue and enqueue can be defined straightforwardly as taking the head of a Pool and appending to the pool respectively. The Pool itself is threaded using the State monad, but is only indexed by demand for cells and of course does not supply any. For completeness we give their signatures:

```
dequeue : ϵ[ M (Emp ∪ Thread) ]
enqueue : ∀[ Thread ⇒ M Emp ]
```

We now focus on the implementation of making a single step in a given thread. To animate a suspended thread we inspect the command in the top-most position of the underlying command tree. We then apply a *handler* which interprets a single command in a given monad and returns a response of the right type. We then apply the continuation with the response which will compute until it again blocks at a command:

```
stepFree : (handler : ∀[ Π[ c ∈ Cmd ]⇒ M (Resp c) ]) →
           ∀[ Free P ⇒ M (Free P) ]
stepFree handler (pure px)              = return (pure px)
stepFree handler (impure (c •⟨ σ ⟩ κ)) = do
  κ •⟨ σ ⟩ r ← κ &⟨ •-comm σ ⟩ handler c
  return (κ ⟨ σ ⟩ r)
```

Normally this would then be used to interpret an entire command tree using a monadic fold. In our scheduler, the thread is put back into the pool after a single step of unfolding.

The implementation of stepFree transparently (and generically) handles all of the resource shuffling. Consider the resources involved in creating a new channel. The command send by the client contains no resources at all, but the response consists of two channel endpoints and thus contains some demand. The state computation returned by the handler in that case encapsulates the response; the new demand will balance out against the new supply. Using tensorial strength we bring the continuation κ into the monadic context and call it with the response. When we call it we intrinsically "transfer" the resources of the argument to the client.

USING THE DEFINITION of stepFree, we reduce the matter of taking a step in a thread to handling each individual command. Let Runtime be an abstract monad for now. In the remainder of this section we focus on the implementation of the commands. These are then dispatched by a function handle:

```
handle : ∀[ Π[ c ∈ Cmd ]⇒ Runtime (Resp c) ]
```

The implementation of step is essentially stepFree handle, lifted to Thread.

THE IMPLEMENTATION OF the command fork *thr* is simply to enqueue the thread *thr*. Resourcewise, we are only shuffling some demand encapsulated in the command tree *thr* to a new slot in the threadpool. We will thus focus on the handling of the more interesting communication primitives: newchan, send, and receive.

Asynchronous communication is implemented using buffered channels, inspired by the buffer threads used by Fowler et al.[41]. The asynchronous, buffered model is a good fit for executable semantics, as it is closer to practical implementations. At the same time it avoids the need to organize a rendezvous between two communicate threads, as all communication is mediated by the buffer.[42]

Type safety of session-typed languages with asynchronous communication relies on a number of invariants related to channels:

1. If a channel endpoint is sending (i.e., has a type $\alpha \,!\, \beta$), then its buffer must be empty.
2. If a channel endpoint is sending then the buffer of the communicating endpoint must accept those values.
3. If one channel has been closed, then the other end *cannot* be sending.
4. If a channel endpoint has been closed, its buffer must be empty.

We will represent the state in a way that makes these observations self-evident from dependent pattern matching.

Buffers are lists of values waiting to be received at an endpoint. We write $\alpha \rightsquigarrow \beta$ (or $\beta \leftsquigarrow \alpha$) for a typed buffer with two ends: the *external* endpoint β corresponding to the endpoint of a channel, and an *internal* endpoint α. The type of the internal endpoint denotes the type of the channel endpoint after it will have caught up with the buffered values:

[41] Fowler et al. 2019. "Exceptional asynchronous session types: Session types without tiers"

[42] The earlier intrinsically typed semantics by (Thiemann 2019) uses synchronous communication and has to find two threads that want to communicate at runtime to make communication happen.

```
data _↜_ : STy → STy → Pred SCtx where
  emp  : (α ↜ α) ϵ
  cons : ∀[ Val a ∗ (β ↜ γ) ⇒ ((a ? β) ↜ γ) ]
```

Channels can then be represented as two linked buffers, or, in case either endpoint has already been closed, a single buffer. The duality between the types of endpoints of a channel holds when both sides are completely caught up with all communication on the channel. Consequently, the duality is enforced at the *internal* endpoints of the buffers:

```
data Channel : Runtype → Pred SCtx where
  dual   : β₂ ≡ β₁ ⁻¹ → ∀[ (α ↜ β₁) ∗ (β₂ ⇝ γ) ⇒ Channel (chanr α γ) ]
  single : ∀[ end ⇝ β ⇒ Channel (endpr β) ]
```

The typing of buffers makes the first invariant hold: we can only have values waiting if the external end is a type a ? β. Whenever we have a buffer whose external endpoint is sending, it is thus self-evident that it is empty and does not contain any resources. Hence, the following is a total definition:

```
sending-buffer-empty : ∀[ (a ! β) ↜ γ ⇒ Emp ]
sending-buffer-empty emp = refl
```

The duality of the internal endpoints of linked buffers then also ensures that the second invariant is satisfied:

```
sending-receiving : ∀[ Channel (chanr (a ! β) γ) ⇒ (γ ↜ (a ? β ⁻¹)) ]
sending-receiving (dual refl (emp •⟨ σ ⟩ b)) rewrite •-id⁻ˡ σ = b
```

The third invariant holds, because the available constructors for buffers of type end ⇝ β restrict β to be either also end, or a ? γ:

```
singlesided-not-sending : ∀ {Φ} → Channel (endpr α) Φ → α ≢ (b ! γ)
singlesided-not-sending (single emp)      ()
singlesided-not-sending (single (cons _)) ()
```

And finally, the fourth invariant:

```
ended-empty : ∀ {Φ} → (end ↜ α) Φ → α ≡ end
ended-empty emp = refl
```

These invariants drive the implementation of pushing and pulling values to and from buffers in a type-safe manner. For example, sending on an given channel is implemented as follows:

Agdaism: We avoid using β^{-1} directly for the type of the right buffer because Agda will then refuse to do case distinction on this buffer unless we force β (so that β^{-1} reduces). Instead we use the standard workaround of generalizing the type and adding an extra argument $\beta_1 \equiv (\beta_2{}^{-1})$. Now we can control whether Agda inlines this equation by pattern matching on refl.

Agdaism: The "absurd" pattern () is used to refute a branch using the fact that the argument has an empty type.

```
push : ∀[ Val a ⇒ (γ ↜ (a ? β)) —∗ (γ ↜ β) ]
push v ⟨ σ1 ⟩ emp = cons (v •⟨ σ1 ⟩ emp)
push v ⟨ σ1 ⟩ cons (w •⟨ σ2 ⟩ b)
  with _ , σ3 , σ4 ← •-assoc_r σ2 (•-comm σ1)
  = cons (w •⟨ σ3 ⟩ (push v ⟨ •-comm σ4 ⟩ b))

send-into : ∀[ Val a ⇒ Channel (chanr α (a ! β)) —∗ Channel (chanr α β) ]
send-into v ⟨ σ ⟩ dual {β1 = x ? β1} refl (b •⟨ σ' ⟩ emp) rewrite •-id⁻ʳ σ' =
  dual refl ((push v ⟨ σ ⟩ b) •⟨ •-id^r ⟩ emp)
```

▲

Figure 4.9: Typed implementation of sending on a channel endpoint. Dependent pattern matching reveals that the linked buffer is capable of receiving the value.

In the definition of send-into we make use of the fact that a channel whose one external endpoint is sending implies that the buffer of the other endpoint will accept that value in its buffer (invariant 2).

The implementation gives a decent impression of what programming with resourceful values in the shallowly embedded logic is like. When working in the logic, it is not necessary to spend time thinking about the "right"/sufficiently general type for these functions with respect to the global invariant that we want to keep. The separation logic connectives encapsulate the proof state so that we can focus primarily on the computational aspects. In the implementation we have to unpack and repack separation witnesses. This is a little tedious, but, importantly, is entirely type-directed and requires no real creativity.

Now that we have an intrinsically typed representation of a channel and the operations to manipulate them, we turn to the communication primitives, which are state operations on lists of channels. The threading of state is taken care of by mixing the State monad that we constructed for linear references into the Runtime monad. The state we thread through is a bunch of channels:

```
Channels = IStar Channel

St : Pred SCtx
St = LeftOver Channels
```

Note that something is slightly different now that we instantiate the Market monoid with session contexts. The demand for two endpoints can be met by the supply of a single channel.

```
example : (○ (Own [ endpr α ] ∗ Own [ endpr β ]) ∗ ● (Own [ chanr α β ])) (supply ϵ)
example = lift (refl •⟨ divide lr •-id^l ⟩ refl)
         •⟨ supply_r •-id^r ⟩
         lift refl
```

The endpoints meet up in the monoid Ends to demand a full channel. This then cancels against the supply of the full channel in the Market monoid.

The specific operations we defined previously for creating, reading, and writing cells are not useful. They each require a channel pointer Own [chanr α], while in practice we will only ever have endpoint pointers Own [endp α]. So rather we implement the following new operations on a monad Runtime:

```
newChan   : ϵ[ Runtime (Own [ endpr α ] ∗ Own [ endpr (α ⁻¹) ]) ]
receive?  : ∀[ Own [ endpr (a ? β) ] ⇒ Runtime (Val a ∗ Own [ endpr β ]) ]
send!     : ∀[ Own [ endpr (a ! β) ] ⇒ Val a —∗ Runtime (Own [ endpr β ]) ]
closeChan : ∀[ Own [ endpr end     ] ⇒ Runtime Emp ]
```

The main difficulty of implementing the communication primitives on existing channels is to find a channel in the state based on a given endpoint reference and update it. We must update it in place because a single endpoint reference does not allow us to take the entire channel out of the supply. Doing so would leave the reference to the other endpoint temporarily dangling. The type of updates that we can then perform safely on the channel is also limited: the endpoint to which we do not have a reference must remain typed as is.

The binding between the endpoint reference and the underlying buffer is established via all the separation witnesses that sit between them. This means that in order to dereference an endpoint reference in the list of channels we must carefully inspect these separation witnesses. Here, we poke through the shallow embedding of our logic to implement the primitive operations on references.

As an example, we show the complete definition of closing a given channel endpoint denoted by a given endpoint reference in figure 4.10. This is the primitive core of the monadic operation closeChan.

```
close'em : (ptr : [ endpr end ] •ₗ ds ≃ xs) → ∀[ Channels xs ⇒ Channels ds ]
close'em (consˡ p) (single emp :⟨ σ ⟩: chs)
  rewrite •-id⁻ˡ p | •-id⁻ˡ σ = chs
close'em (consʳ p) (ch :⟨ σ ⟩: chs) = ch :⟨ σ ⟩: close'em p chs
close'em (divide lr p) (dual refl (emp •⟨ τ ⟩ bᵣ) :⟨ σ ⟩: chs)
  rewrite •-id⁻ˡ p | •-id⁻ˡ τ        = single bᵣ :⟨ σ ⟩: chs
close'em (divide rl p) (dual eq (bₗ •⟨ τ ⟩ emp) :⟨ σ ⟩: chs)
  rewrite •-id⁻ˡ p | •-id⁻ʳ τ
  rewrite dual-neutral eq            = single bₗ :⟨ σ ⟩: chs
```

▲

Figure 4.10: The implementation of the primitive for closing a channel endpoint. We write $x :\langle\ \sigma\ \rangle: xs$ for cons $(x \bullet\langle\ \sigma\ \rangle\ xs)$.

The type of the operation close'em already pokes through the state abstractions. For the representation of *ptr* we use the separation witness that we obtained from a conjunction in the market monoid:

$$\bigcirc\ (\text{Own [endpr end]}) * \bullet\ \text{St}.$$

In the signature of close'em we already unpacked both modalities and 'manually' enforce that the endpoint reference can be dereferenced in the channel state using the index *xs*.

We can also see here how the dynamic binding is resolved. The representation of the *ptr* is literally pointing towards the location of the endpoint in the channels.[43] The constructor cons^l says that it is the head of the channel list that we are looking for. The type of the pointer enforces that this must then be a single buffer channel without values in it. We simply discard it, which is resource neutral by the identity law of the separation logic. If instead we find the constructor cons^r we continue following the 'remainder of the pointer' on the tail. The branch divide *lr* (respectively divide *rl*) corresponds to the case where the endpoint is the left (respectively the right) endpoint of a dual channel in the head of the channel list.

To SUMMARIZE, we have defined the typed representation for a concurrent runtime with asynchronous, buffered communication. Although the typing of the runtime for a session-typed language contains some subtleties, the staged interpretation of this language required no changes at the level of the logic. The logic and monadic abstractions that we defined scale to the functional session-typed language LTLC_{ses} and indeed yield interpreters whose clarity is not obscured by explicit proof terms.

[43] That is, our pointers take the shape of the data that they point into. This is the more general lesson taught by McBride by his work on the co-de-Bruijn representation.

4.10 Related Work

We discuss prior work on using dependent types to express linear and relevant scoping and typing of terms, and to prove safety of session-typed languages[44,45,46]. We also discuss the work on separation logic that was used to develop PRSAs.

Mechanized Metatheory of Session-Typed Languages. In the last few years there has been an increase in efforts to mechanize the metatheory of process calculi[47] and also session-typed functional languages. For example, recently Castro-Perez et al.[48] developed tools in the Coq proof assistant to aid in the mechanized verification of subject reduction for the calculus originally proposed by Honda et al.[49], as well as for the revised language by Yoshida et al.[50]. Zalakain et al.[51] mechanize the same theorem in Agda for a resource-aware pi-calculus using different techniques—most notably *leftover typing*[52].

All of these approaches follows a conventional extrinsic approach to typing and proofs. The most closely related work on mechanizing session-types metatheory is an *intrinsically-typed* small-step operational semantics in Agda by Thiemann[53]. The session-typed functional language that he uses is a superset of ours. It includes internal and external choice operators (with subtyping), a coinductive definition of session types, and unrestricted types. He gives a round-robin scheduler for threads, and implements synchronous communication. The semantics takes the form of an interruptible abstract machine[54], that operates by decomposing an expression with its value environment and evaluation context, into a command for the scheduler. In addition, he proves that his semantics implements the beta rule for unrestricted function application, and an eta rule for pairs. The complete development (minus import statements) comprises 2890 lines of Agda code. Of those lines, 1652 are used to define the semantics.

The cited work presents a mechanized semantics of a session-typed language in an executable, intrinsically typed style, and identifies the role of both static and dynamic separation in the type-safety proof. One of the key issues that Thiemann identifies, which directly provoked present work, is the difficulty of managing the separation of the resources. Particularly, the pervasiveness of proof terms related to the separation of resources make many of the def-

[44] Takeuchi et al. 1994. "An Interaction-based Language and its Typing System"

[45] Honda et al. 1998. "Language Primitives and Type Discipline for Structured Communication-Based Programming"

[46] Vasconcelos et al. 2006. "Type checking a multithreaded functional language with session types"

[47] Milner et al. 1992. "A Calculus of Mobile Processes, I"

[48] Castro-Perez et al. 2020. "EMTST: Engineering the Meta-theory of Session Types"

[49] Honda et al. 1998. "Language Primitives and Type Discipline for Structured Communication-Based Programming"

[50] Yoshida et al. 2007. "Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication"

[51] Zalakain et al. 2021. "π with Leftovers: A Mechanisation in Agda"

[52] Allais 2017. "Typing with Leftovers - A mechanization of Intuitionistic Multiplicative-Additive Linear Logic"

[53] Thiemann 2019. "Intrinsically-typed mechanized semantics for session types"

[54] Felleisen et al. 1987. "Control operators, the SECD-machine, and the λ-calculus"

initions "tedious exercise[s] in resource shuffling". This includes important definitions for the semantics, such as the function that decomposes expressions. Another example is the function that searches in the thread pool for the send command that corresponds to a read command. This function comprises 50 lines of Agda and its type quantifies over 7 contexts, related by 3 separation witnesses. The interesting case of this function, which considers a send command, requires 11 lines of code, 10 of which are reorganizing the six separation terms that appear in the arguments of function.

We address the complexity and tedium of working with separated objects by composable abstractions. Each of the abstractions can be understood separately, making the complexity of the composite manageable. An important part of this is that we are able to give recognizable types to abstractions by working in an appropriate logic. Additionally, the abstractions help considerably to avoid duplication. For example, we are able to prove separation rotation lemmas for all PRSAs using just •-assoc and •-comm, whereas they are proven separately for both separation of typing contexts and session contexts by Thiemann. Another good example is his definition of session context separation, which has 6 constructors, baking in the separation on channels. Using our library, one can define it as the composition of two PRSAs, reusing our instance for lists. Besides the reuse one gets, this also simplifies the proofs.

Besides the proof terms, the styles of the semantics also differ significantly. Whereas a small-step semantics really shines in a relational setting, it is an indirect way to define an executable semantics. The computation steps of the language are hidden within the mechanics of the abstract machine—i.e., in decomposing expressions, and plugging values back into evaluation contexts. Some of the clarity is lost in these indirections that animate the small-step semantics. In our definitional interpreter, this tension between the small-step nature of concurrency and the functional style that we require, is resolved by using the free monad, representing continuations not as syntax, but as Agda functions. Additionally we avoid searching in evaluation contexts to find two threads that are both ready to communicate, by mediating between communicating threads using a buffer.

Another *intrinsic* approach to linear metatheory is proposed by Wood et al.[55], using linear algebra over semirings. The abstrac-

[55] Wood et al. 2020. "A Linear Algebra Approach to Linear Metatheory"

tion over an arbitrary semiring enables the generalization from strictly linear languages to language with more lax sub-structural constraints. Their approach yields generic definitions and properties for substitution and renaming for this class of languages.

Co-de-Bruijn Representation of Syntax. The term co-de-Bruijn was coined by McBride[56], who exploits this representation in Agda to ensure that variable shifts in well-scoped, non-linear lambda terms do not require term traversals. He shows that hereditary substitutions on these terms become structurally recursive. Rather than working directly with objects in their relevant context, he works with objects wrapped in "thinnings" into a larger context, and develops the structure of these wrapped objects. He defines "relevant pairs", which are almost identical to our separating conjunctions for typing contexts, but permit overlap between the consumption of the left and right projections.

[56] McBride 2018. "Everybody's got to be somewhere"

Abel et al.[57] describe an encoding of the binding in the simply-typed lambda calculus terms that is inspired by the typing of terms in a linearly-typed lambda calculus. They adapt it to non-linear lambda calculi and use it to avoid space leaks in interpreters that build function closures. This is achieved by separating environments along context separations in the interpreter, thus only capturing relevant values and avoiding leaks. This separation of the environment reappears in our linear interpreters in the generic frame operation of the Reader monad.

[57] Abel et al. 2011. "A Lambda Term Representation Inspired by Linear Ordered Logic"

Neither Abel et al.[58], nor McBride[59] make the connection with separation logic, or make use of a magic wand-like connective, which was crucial in typing our functional abstractions.

[58] Abel et al. 2011. "A Lambda Term Representation Inspired by Linear Ordered Logic"

[59] McBride 2018. "Everybody's got to be somewhere"

Separation Algebras. In the development of our abstractions and interpreters we have found a lot of inspiration in existing work on separation logic. We have already mentioned the work that contributed to the formulation of PRSAs in section 4.4. For constructing the logic on top of PRSAs we were directly inspired by the construction of separation logic in Iris[60], which defines the type formers and connectives of separation logic in terms of *cameras*—i.e., the variant of separation algebras used in Iris.

[60] Jung et al. 2018b. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic"

Partial (commutative) semigroups and monoids make their appearances in more domains than separation logic. In type systems, a noteworthy appearance is the formalization of *row types*. Morris et

al.[61] present (commutative) row algebras to generalize and abstract languages with row types, and also point to many instances of their algebras in other work. Commutative row algebras are similar to separation algebras of Dockins et al. (2009), but use a partial function to characterize the operation of a partial monoid. In prior work, the choice of using a propositional relation versus using a partial function for the monoidal operation mainly depends on what suits the meta-language in which one develops the theory. In our work, we cannot use either because monoidal operations such as $\_\bullet_l\_\simeq\_$ in section 4.8 are non-deterministic (or, not functional). Sometimes this can be dealt with by quotienting the carrier of the monoid by an appropriate equivalence and defining a partially deterministic monoid on the quotient.[62] In our work, we not only have a non-deterministic monoid (i.e., different compositions of the same elements: $a \bullet b \simeq c$ and $a \bullet b \simeq d$ where $c \neq d$), but also a *proof relevant* monoid (i.e., $a \bullet b \simeq c$ in multiple, essentially different ways).

The Market PRSA is inspired by Iris's Auth camera[63], which serves a similar purpose: relating the provider of a resource to its clients. Elements of Auth A are essentially pairs $(x\ ,\ y)$ of A's. The left *authoritative* element can either be present or not, and cannot be separated. If it is present, then we must have the inclusion $x \bullet z \simeq y$, for some leftover resource z. Unfortunately, because of the inclusion evidence, the Auth construction does not transfer well to the proof-relevant setting: to prove the laws of Auth A for an arbitrary PRSA A, one needs the higher structure of A—e.g., to prove that composing $\bullet\text{-assoc}_r$ with $\bullet\text{-assoc}_l$ is the identity. The Market PRSA is a generalization of the model for counting permissions by Bornat et al.[64].

Linear Types and Separation Logic. We used ideas from separation logic to write interpreters for linear languages that are intrinsically type safe. There has also been prior work on using separation logic to prove type safety in an extrinsic manner. A notable development in this direction is RustBelt[65], where they used the technique of *logical relations* in Iris[66] to prove type safety and data race freedom of the Rust type system and some of its standard libraries. Contrary to our work, these developments on logical relations are based on an untyped operational semantics instead of a well-typed interpreter.

Another line of recent work developed a separation logic for proving functional correctness of message-passing programs called Actris[67]. Actris has a notion of dependent protocols inspired by

[61] Morris et al. 2019. "Abstracting extensible data types: or, rows by any other name"

[62] Krishna et al. 2018. "Go with the flow: compositional abstractions for concurrent data structures"

[63] Jung et al. 2015. "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning"

[64] Bornat et al. 2005. "Permission accounting in separation logic"

[65] Jung et al. 2018a. "RustBelt: Securing the foundations of the Rust programming language"

[66] Krebbers et al. 2017. "Interactive proofs in higher-order concurrent separation logic"

[67] Hinrichsen et al. 2020. "Actris: Session-type based reasoning in separation logic"

session types. In recent work, Actris was used to prove semantic soundness of a small-step operational semantics for a language with session-typed communication in conjunction with other features such as polymorphism.[68]

[68] Hinrichsen et al. 2021. "Machine-checked semantic session typing"

4.11 Conclusions

We presented the development of intrinsically-typed interpreters for LTLC_{ref} and a session-typed language LTLC_{ses}. The Agda development consists of (1) the library for proof-relevant separation logic and the described functional abstractions, (2) an interpreter for LTLC and LTLC_{ref}, and (3) the syntax and semantics of LTLC_{ses}.

The interpreters that we defined for LTLC_{ref} (section 4.7) and LTLC_{ses} (section 4.8) are executable and type-safe specifications whose meaning is not obscured by explicit proof work. The few proof terms that are still present in their definitions are trivial to fulfill using the laws of PRSAs, and thus impact the clarity of the semantics less than the partiality present in untyped interpreters. Using tensorial strength to enable programming with the external bind of these monads, and using a free monad to implement concurrency and communication, we managed to preserve the familiar look of monadic interpreters.

In the present work we focused on the typing and use of a monadic interface for effects in the presence of separation. The *implementations* of the monadic operations were of lesser concern, because they are generic and reusable. These operations have to poke through the abstractions of its interface (i.e., the logic) and look at the separation witnesses. It would be interesting to further investigate if this can be improved. It seems that one could avoid some more manipulation of the separation witnesses if one adopts a completely point-free programming style. In current day dependently-typed languages this is not very appealing, because we cannot use the builtin support for defining functions using dependent pattern matching.

Because the remaining proofs of separation are mechanical, using only the axioms of the PRSAs, it seems likely that we can use some lightweight automation to fill them in.[69] We would also like to investigate whether Agda's rewriting[70,71] could be used to automatically eliminate separation witnesses containing ϵ.

[69] See, for example, Krebbers 2015, §9.6.

[70] Cockx et al. 2016. "Sprinkles of extensionality for your vanilla type theory"

[71] Cockx 2020. "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules"

5 Typed Compilation with Nameless Labels

"If you want to go somewhere,
goto is the best way to get there."
— Ken Thompson

In chapters 3 and 4, we looked at interpreters for different programming languages. In this section, we look at another programming language backend: *compilers*. Compilers are usually implemented as a pipeline of *transformations* of a source program. Each transformation brings the source program closer to an optimized program in the—typically low-level—target language. In this chapter we develop a compiler pipeline that transforms programs from a small imperative language IMP[1] to programs in a subset of JVM bytecode[2]. The multi-stage compiler pipeline is depicted on the right. This chapter will mainly focus on the third stage, which does the translation of structured control flow (e.g., if-then-else expressions and while-loops) to bytecode with jumps to labeled instructions.

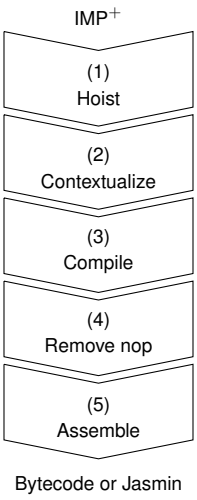
Formally, we can understand compiler transformations as translations between languages. The source, intermediate, and target languages can each be equipped with their own static and/or dynamic semantics. Each transformation then poses verification challenges. Compilers that go wrong turn "correct" source programs into "incorrect" target programs. By relating the semantics of the source and target languages of transformations, we can make that statement more precise: compilers that go wrong translate source programs to target programs with an unrelated semantics. Conversely, a correctness criterion for a compiler transformation states that it translates between programs whose semantics are related.

There are different correctness criteria that we can consider, depending on the type of semantics for the languages and the different

The technical material of this chapter was published in A. Rouvoet, R. Krebbers, and E. Visser (2021). "Intrinsically-typed compilation with nameless labels". In: *Proceedings of the ACM on Programming Languages* 5.POPL, pp. 1–28. DOI: 10.1145/3434303.

[1] Nipkow et al. 2014. "IMP: A Simple Imperative Language"

[2] Lindholm et al. 2020. "The Java Virtual Machine specification: Java SE 14 edition"



notions of relatedness between them. *Functional correctness* of compilers is a very strong result proving a simulation relation between the dynamic semantics of the source and the target of compilation. The most extensive and well-known projects in this direction are CompCert[3] and CakeML[4], which provide fully functionally verified compilers for the C and ML programming languages, respectively. The great confidence in such compilers comes at the price of the research and development effort that is required to state and establish their correctness. Projects like CompCert and CakeML are the result of a decade of work into specifying the semantics of the (intermediate) languages involved in the compiler, and specifying and proving the simulations between these semantics. If we want to avoid these costs of functional verification, but still want to avoid a large class of miscompilations, we can instead verify *type correctness* of a compiler[5,6,7,8]. This is a weaker property than functional correctness, but is much easier to specify and prove, as it does not involve the dynamic semantics of the source and target languages of transformations. At the same time, it still rules out many bugs in a compiler, because we are guaranteed that well-typed target code does not "go wrong"[9].

[3] Leroy 2009. "Formal verification of a realistic compiler"

[4] Kumar et al. 2014. "CakeML: A verified implementation of ML"

[5] Chlipala 2007. "A certified type-preserving compiler from lambda calculus to assembly language"

[6] Guillemette et al. 2008. "A type-preserving compiler in Haskell"

[7] Bélanger et al. 2015. "Programming type-safe transformations using higher-order abstract syntax"

[8] Abel 2020. "Type-preserving compilation via dependently typed syntax in Agda"

[9] Milner 1978. "A theory of type polymorphism in programming"

One method to verify type correctness of a compiler is by implementing it in a proof assistant and proving the type correctness property *extrinsically*—i.e., by writing a separate proof that reasons about the implementation of the compiler. This approach has the same disadvantages as extrinsically proving type-safety of interpreters. First, verification may be difficult if one does not implement the compiler in way that accommodates verification. Additionally, one only discovers bugs by failing to finish the type correctness proof, so that feedback on the design of the compiler comes at a late stage in the development rather than interactively during the implementation.

As part of our mission to equip language developers with the tools to implement typed languages safely, we believe that it should be feasible to implement compilers in an intrinsically typed fashion instead. That is, to define data types representing well-typed source *and* target programs, and to implement the compiler as a function from the former to the latter. We call this an *intrinsically-typed*[10] compiler[11], because it verifies the type-correctness property of the compiler internally, as part of the definition. Typed language trans-

[10] Reynolds 2000. "The meaning of types from intrinsic to extrinsic semantics"

[11] McKinna et al. 2006. "A type-correct, stack-safe, provably correct, expression compiler"

formations tend to be more complicated than typed interpreters, because the output of a transformation is an entire program, rather than a value. The type invariants of programs tend to be more complicated than those of values. For example, because programs have more forms of name binding. In this chapter we extend the state of the art of intrinsically typed compilation to compilers that output a *linear* representation of bytecode—i.e., where all control-flow is represented by means of jump instructions to labeled code. Producing linear code is an essential step of a complete compiler.

5.1 The Problem of Intrinsically-Typed Labels and Jumps

Consider the following compilation of a conditional expression.

```
Compiler : Set → Set
Compiler A = Label → Label × Bytecode × A

compile : Exp → Compiler ⊤
compile (if c then e₁ else e₂) = do
  compile c
  lthen ← freshLabel
  tell [ iftrue lthen ]
  compile e₂
  lend ← freshLabel
  tell [ goto lend ]
  attach lthen (compile e₁)
  attach lend  (return tt)
```

Pre stack type	Label	Instruction(s)	Post stack type
ψ		⟦ c ⟧	boolean :: ψ
boolean :: ψ		iftrue *lthen*	ψ
ψ		⟦ e_2 ⟧	a :: ψ
a :: ψ		goto *lend*	a :: ψ
ψ	*lthen*	⟦ e_1 ⟧	a :: ψ
a :: ψ	*lend*	nop	a :: ψ

▲

Figure 5.1: A single case of a monadic compiler, and a table with the corresponding compilation template. The expression c is a boolean and e_1 and e_2 are typed a. We use ⟦ e ⟧ to denote the compilation of e.

It is written in a declarative style that exactly mirrors the compilation template shown on the right. Unfortunately, even if we implement this compiler in a typed language like Haskell, the programmer can still make mistakes that are essentially type errors. For example, the type checker cannot enforce that the programmer attaches the label *lthen* after using it as a jump target. If the programmer fails to do this, then the output of the compiler is not well-bound (and thus also not well-typed) bytecode. Ideally, we want to rule out all such mistakes and guarantee that any compilation function that is type-correct in the host language, outputs well-typed bytecode.

To rule out these mistakes, we can try to make use of an *intrinsically-typed* representation of bytecode. Using dependent types, we can

strengthen the types of the embedding of bytecode, as well as the (monadic) operations that manipulate it, so that these express the type constraints of bytecode. The *goal* of this chapter is to accomplish exactly that. Importantly, we want to do this in such a way that we can use these new and improved operations to define compile without spoiling its declarative nature. Key to this is ensuring that the formulation of bytecode typing is *compositional*, so that the well-typed operations (like the monadic operations) can be composed into a well-typed compiler with little manual proof effort.

The declarative nature of the untyped compiler above is accomplished by writing it as a computation in the monad Compiler, which combines a writer monad with state. The writer part collects the generated instructions, and the state part provides a supply of fresh labels. The (writer-) monad operation tell is used to output a *given* sequence of instructions, and the operation attach is used to label the first instruction of the output of a compiler computation. Recursive invocations of the compiler are used to produce the output for the sub-expressions.

Assuming that the input expression is well-typed, we now ask the question: is the result of the compiler well-typed? The typing of instructions is based on their effect on the state of the machine that executes them. For the set of instructions shown, the part of the state that matters is the operand stack. Type correctness of the bytecode produced by the compiler requires the stack types to match between instructions and subsequent instructions. For most instructions this is the next instruction in the sequence, but for (conditional) jumps (i.e., iftrue and goto) this is an instruction marked by a label. In addition, labels must be *well-bound*—i.e., they must unambiguously reference an instruction in the output. Concretely this means that labels must be declared *exactly once*.

To see that the compilation of conditional expressions in figure 5.1 is indeed type correct, we need to make use of the *stack invariant* of expression compilation, which states that executing the bytecode for an expression e of type a in an initial stack configuration typed ψ will leave a single value of type a behind on top of the initial stack ψ. Using this invariant we can construct the typing for the bytecode template shown in the table in figure 5.1, and verify that it satisfies the type-correctness criterion. For example, iftrue *lthen* is followed either by the subsequent instructions for the "else" branch, or the bytecode labeled *lthen* for the "then" branch. Both expect a stack

typed ψ, matching the type of the stack after popping the condition.

In verifying this correctness argument intrinsically, we encounter two conceptual problems:

- Many operations of the compiler are simply not well-bound in isolation. For example, the third operation tell [iftrue *lthen*] in the compiler in figure 5.1, which outputs the iftrue instruction, produces a jump with a dangling label reference *lthen*. That is, it is type correct only provided that the label *lthen* is eventually attached to some bytecode in the output.
- Even though the compilation in figure 5.1 appears locally well-bound, type correctness crucially depends on the generated labels *lthen* and *lend* being *fresh* for the *entire* compilation. That is, type correctness depends not only on which *concrete* labels we fill in for the symbolic labels in the template, but also on whether those are different from the labels used in the surrounding bytecode, and the bytecode of the condition and branch expressions.

In other words, bytecode typing appears to be a whole-program property, that is simply not preserved by the individual operations of our compiler. This makes it very difficult to intrinsically type the operations of compilation. As we show in section 5.2, attempts to do this result in instrinsically-typed compilers that are bloated with explicit proofs of side conditions. This puts a burden on the programmer to prove these side conditions, and spoils the declarative nature of the compiler.

Key idea 1: Nameless co-contextual bytecode. If bytecode typing is indeed inherently anti-compositional due to global binding, how can we expect to obtain a type-correct compiler by mere composition of the individual operations? We identify two reasons that make bytecode and its typing anti-compositional: (1) the *contextual* formulation of traditional typing rules, and (2) the use of names to represent global label binding in bytecode terms. To address these issues, we propose a new *nameless* and *co-contextual*[12] typing of bytecode.

In traditional typing rules, one deals with bound names (like labels) using contexts. Contexts are inherently non-local, as they summarize the *surroundings* of a term. In bytecode, the scope of labels is not restricted, and consequently the label context contains *all* labels in the program. This forces us into taking a whole-program

[12] Erdweg et al. 2015. "A co-contextual formulation of type rules and its application to incremental type checking"

perspective. For example, concatenating two independent, contextually typed bytecode fragments requires *extrinsic* evidence that the defined labels are disjoint. This requires non-local reasoning, for example, using a supply of provably, globally fresh label names. It also requires weakening of typed bytecode as we bring more labels into scope.

The co-contexts that we use on the other hand, only describe the types of the provided label *exports* (i.e., label binders) and the required label *imports* (i.e., the label references in jumps) of a bytecode fragment. Co-contexts are *principal*[13], in the sense that they are the smallest set of exports and imports that work. This means that weakening is never required. Moreover, by using a nameless representation, there cannot be accidental overlap between the exported labels. This means that we can *always* concatenate bytecode fragments, simply adding up exports and imports. Our representation is inspired by the nameless co-de-Bruijn representation of lexical binding by McBride[14], and the nameless representation of linear references in chapter 4.

[13] Jim 1996. "What are principal typings and what are they good for?"

[14] McBride 2018. "Everybody's got to be somewhere"

Key idea 2: Programming with co-contexts using separation logic. The flipside of the co-contextual and nameless typed representation of bytecode, is that a lot of information is encapsulated in a single co-context composition relation. This proof-relevant relation appears everywhere and is hard to manipulate by hand. To avoid this, we show that this relation forms a proof-relevant separation algebra (see section 4.4). The induced shallow embedding of separation logic abstracts over co-contexts and their compositions, enabling us to implement an intrinsically-typed version of the compiler in figure 5.1 with little manual proof work. The key result of our separation logic is the intrinsically-typed freshLabel operation:

```
freshLabel : (Binder ψ ∗ Reference ψ) ϵ
```

Because labels are nameless, there is no need for state, and thus freshLabel is not monadic. To ensure well-boundness, we distinguish the two roles of labels in the types: *binding* and *reference* occurrences. The operation freshLabel constructs a pair of both a binding occurrence of type Binder ψ and a reference occurrence of type Reference ψ of the *same* label. The stack type ψ ensures that the two occurrence are used in type-compatible positions. The two occurrences are paired using a *separating conjunction* $*$. This hides the co-contexts, as

well as the composition of these co-contexts that binds the reference occurrence to the binding occurrence. The ϵ on the outside is the *empty co-context*. The fact that the pair is typed using the empty co-context can be understood as this pair being internally well-bound: it does not import any labels, nor export any labels for a user on the outside (i.e., the generated label really is fresh).

The rules of co-context composition prohibit the binding occurrence from being duplicated or discarded. This means that to the programmer, the binding occurrence returned by freshLabel behaves like a *linear value*. It can be passed around freely, but must eventually appear in the output of the compiler. This ensures that jumps in the output are intrinsically well-bound.

Technical contributions In this chapter we present an approach to *intrinsic* verification of type-correctness of compilers. After discussing the key challenges and the key ideas that we use to overcome these challenges (section 5.2), we make the following technical contributions:

- In section 5.3 we present a new nameless and co-contextual representation of typed global label binding. We formalize it using a generic, *proof-relevant*, ternary composition relation, that characterizes exports and imports and their possible interactions.

- In section 5.5 we prove that this relation forms a proof-relevant separation algebra (PRSA)—i.e., it is commutative, associative, and has a unit. This again yields a separation logic that abstracts over co-contexts and their compositions. This logic provides a high-level language for writing intrinsically-typed programs that use nameless labels.

- In section 5.7 we present a co-contextual typing of a subset of JVM bytecode with labels and jumps. We use our separation logic as a framework for specifying the typing rules, without explicitly talking about co-contexts and their compositions. We show that our co-contextual representation can be translated into code with absolute jumps instead of labels.

- In section 5.8 we present the compilation of a small expression language with structured control-flow to JVM bytecode. To implement the compiler at the right level of abstraction with little

manual proof work, we develop a linear writer monad for compiling with labels, implemented on top of our separation logic.

- In section 5.9 we explain how the compilation pass fits in an intrinsically-typed compiler backend that we implemented in Agda. The backend also includes a source language transformation (local variable hoisting), a target language optimization (noop removal), and a transformation that eliminates labels in favor of instruction addresses.

We finish the chapter with a discussion of related work (section 5.10), and a conclusion (section 5.11).

5.2 *Intrinsically Verifying Label Well-Boundness*

In this section we show the straightforward named and contextual typing of bytecode, and why it falls short for intrinsically typed compilation. We then explain our key ideas: nameless and co-contextual typing of bytecode, and using separation logic to program with typed bytecode.

BYTECODE WITH LABELS can be described as a simple term language

labels	ℓ		
constants	k		
plain instructions	i	:=	pop | push k | goto ℓ | ...
labeled code points	c	:=	plain i | labeled ℓ i
bytecode	b	:=	nil | cons c b

The plain instructions of this language can be typed using the judgment $I \vdash i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle$, where the stack types (lists of types) ψ_1 and ψ_2 express a precondition on the entry stack, and a postcondition to the next instruction in the bytecode sequence, respectively. The map I gives the entry stack typing for labeled instructions, so that jumps (goto) can be typed:

Figure 5.2: Contextually- and extrinsically-typed instructions.

▼

$$\text{T-Pop}\ \frac{}{I \vdash \text{pop} : \langle a :: \psi \rightsquigarrow \psi \rangle} \qquad \text{T-Push}\ \frac{k : a}{I \vdash \text{push}\ k : \langle \psi \rightsquigarrow a :: \psi \rangle} \qquad \text{T-Goto}\ \frac{I(\ell) = \psi_1}{I \vdash \text{goto}\ \ell : \langle \psi_1 \rightsquigarrow \psi_2 \rangle}$$

The postcondition ψ_2 of goto is unconstrained because the next instruction in the sequence will not be executed after the goto has

performed the jump. Instead, there is a premise $I(\ell) = \psi_1$ that ensures that the labeled target instruction ℓ—i.e., the actual next instruction of goto at runtime—has the right entry stack type.

Unlike plain instructions, code points can define labels. The judgments $E, I \vdash c : \langle \psi_1 \rightsquigarrow \psi_2 \rangle$ for code points and $E, I \vdash b : \langle \psi_1 \rightsquigarrow \psi_2 \rangle$ for bytecode therefore use an additional context E, which maps each binding occurrence of a label to its entry stack type. This context is *linear* so as to enforce that every label is defined *exactly once*:

Figure 5.3: Contextually- and extrinsically-typed bytecode.

▼

$$\textsc{T-Plain}\ \frac{I \vdash i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \qquad E = \emptyset}{E, I \vdash \mathsf{plain}\ i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle} \qquad \textsc{T-Labeled}\ \frac{I \vdash i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \qquad E = \ell \mapsto \psi_1}{E, I \vdash \mathsf{labeled}\ \ell\ i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle}$$

$$\textsc{T-Nil}\ \frac{E = \emptyset}{E, I \vdash \mathsf{nil} : \langle \psi \rightsquigarrow \psi \rangle} \qquad \textsc{T-Cons}\ \frac{E_1, I \vdash c : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \qquad E_1 \sqcup E_2 \simeq E \qquad E_2, I \vdash b : \langle \psi_2 \rightsquigarrow \psi_3 \rangle}{E, I \vdash \mathsf{cons}\ c\ b : \langle \psi_1 \rightsquigarrow \psi_3 \rangle}$$

The linear behavior of the context E is visible in the leaves (T-Plain, T-Labeled, T-Nil), where we restrict E to the smallest possible context (i.e., no weakening), and in the nodes (T-Cons), where the condition $E_1 \sqcup E_2 \simeq E$ *separates* E in disjoint fashion among the sub-derivations (i.e., no contraction)[15]. We refer to the disjoint separation of the linear context as the *disjointness condition* of label well-boundness.

[15] Walker 2005. "Substructural type systems"

The typing judgment $E, I \vdash b : \langle \psi_1 \rightsquigarrow \psi_2 \rangle$ itself does not enforce that every referenced label has a corresponding binder. Hence, in addition to a proof of the typing judgment, an *inclusion condition* $I \subseteq E$ needs to be proven at the top-level for the whole bytecode program. Equivalently, it suffices to prove a judgment $E, E \vdash b : \langle \psi_1 \rightsquigarrow \psi_2 \rangle$ at the top level. After all, when $I \subseteq E$ holds, we can *weaken* a typing $E, I \vdash b : \langle \psi_1 \rightsquigarrow \psi_2 \rangle$ to $E, E \vdash b : \langle \psi_1 \rightsquigarrow \psi_2 \rangle$.

The first step towards proving compiler type-correctness *intrinsically* is to internalize the typing rules in the bytecode syntax. The result is an inductive type family Bytecode $\psi_1\ \psi_2\ E\ I$, indexed by the two stack types ψ_1 and ψ_2 of type StackTy[16], and the maps E and I of type LabelCtx:

[16] McKinna et al. 2006. "A type-correct, stack-safe, provably correct, expression compiler"

```
StackTy  = List Ty
LabelCtx = Map Label StackTy
```

For brevity we do not specify the set of types Ty here, and we leave the type of maps Map abstract. The typed syntax is as follows:

```
data Instr : StackTy → StackTy → LabelCtx → Set where
  pop    :                Instr (a :: ψ) ψ I
  push   : Constant a →   Instr ψ (a :: ψ) I
  goto   : I [ ℓ ]= ψ1 →  Instr ψ1 ψ2 I

data Code : StackTy → StackTy → LabelCtx → LabelCtx → Set where
  plain   :         Instr ψ1 ψ2 I → Code ψ1 ψ2 ∅ I
  labeled : ∀ ℓ →   Instr ψ1 ψ2 I → Code ψ1 ψ2 (ℓ ↦ ψ1) I

data Bytecode : StackTy → StackTy → LabelCtx → LabelCtx → Set where
  nil  : Bytecode ψ1 ψ1 ∅ I
  cons : Code ψ1 ψ2 E1 I → (E1 ⊔ E2 ≃ E) → Bytecode ψ2 ψ3 E2 I → Bytecode ψ1 ψ3 E I
```

▲

Figure 5.4: Contextually- and intrinsically typed bytecode

Intrinsically typed label references are treated in a traditional manner: context membership $I\ [\ \ell\ ]{=}\ \psi_1$ says that context I contains the label ℓ of stack type ψ_1. Label definitions are also treated in the same way as in the typing rules of figure 5.3 using the linear context E. Consequently, constructor plain sets E to be the empty map $\emptyset$, and the constructor labeled sets E to be the singleton map $\ell \mapsto \psi_1$.

Again, we point out the side conditions that one needs to prove to ensure well-boundness. The disjointness condition $E_1 \sqcup E_2 \simeq E$ is intrinsic to the constructor cons of Bytecode and ensures that labels are bound at most once. To ensure that every label that is referenced is also bound, it is necessary to extrinsically prove the top-level inclusion condition $I \subseteq E$ for the whole program. We now show that these side conditions are a key obstacle that prevent us from implementing an intrinsically typed compiler without manual proof work.

LET US TAKE A LOOK at a candidate signature for an intrinsically typed counterpart to the untyped monadic compiler in figure 5.1:

```
compile : Exp a → ∀ E1
        → ∃⟨ (λ E2 → (E1 # E2) × Bytecode ψ (a :: ψ) E2 E2) ⟩
```

Here, Exp a is a type family for well-typed source expressions of type a. Similar to the untyped compiler, we thread a supply of fresh labels, represented concretely using the label context E_1 of already defined labels. The compiler then returns the set of newly

introduced labels E_2, which must be disjoint from E_1. We write disjointness E_1 # E_2, which is defined as $\exists\langle\ (\lambda E \to E_1 \sqcup E_2 \simeq E)\ \rangle$. The returned bytecode Bytecode ψ $(a :: \psi)$ E_2 E_2 binds all the labels of E_2 and thus satisfies the inclusion condition. The stack indices ψ (for the pre stack type) and $a :: \psi$ post stack type enforce the stack invariant of compiled expressions.

Although it is possible to give a direct implementation of the function compile with the given signature, such implementations do not have the declarative appeal of the untyped compiler in figure 5.1. Compilation of expressions like if c then e_1 then e_2 requires multiple recursive invocations of the compiler, which requires threading of the label context, as well as manual concatenation of the outputs. The latter comes with proof obligations. In particular, we have to weaken the imports of typed bytecode to the set of all generated labels. We also have to prove the disjointness conditions required for concatenation. This results in a definition that is bloated with proof terms.

One may wonder if we can hide the threading of the label context, and the concatenation of the output in a monad, as we did in the untyped compiler. This would require us to type the monadic operations (return, bind, freshLabel, tell, and attach) so that we can implement the compiler compositionally with little manual proof work. In what follows, we will argue that contextually typed bytecode is not well-suited for that, because the two conditions for label well-boundness are inherently anti-compositional, whole program properties.

THE FIRST PROBLEM is that the *inclusion condition* of bytecode only holds for whole programs. That is, while it holds for the output of compile, it does not hold for the individual monadic operations that we wish to use. For example, the output written by tell [goto l], which has type Bytecode ψ ψ $\emptyset$ $(\ell \mapsto \psi)$, does not satisfy the inclusion condition, because $(\ell \mapsto \psi) \not\subseteq \emptyset$. It thus appears inevitable that the inclusion condition is proven *extrinsically* in the implementation of expression compilation, because it is something that only holds at the level of whole programs, and not at the level of the individual monadic operations.[17]

THE SECOND PROBLEM is related to the *disjointness condition* of well-boundness. In the untyped compiler in figure 5.1, disjointness is

[17] The simple appearance of the top-level condition can be misleading. In most compilations the maps I and E are *not concrete*, so that a proof requires symbolic reasoning.

morally ensured through principled use of freshLabel. An intrinsically typed version of freshLabel will thus have to provide some evidence of freshness: a proof $(\ell \mapsto \psi)$ # E_1 that the returned label ℓ is disjoint from already bound ones E_1. This evidence is required when invoking attach ℓ. The problem, however, is that the freshness evidence is not stable. Consider, for example, the following untyped compilation:

```
do
  ℓ ← freshLabel
  compile e
  attach ℓ (return tt)
```

Assume it is indeed the case that before generating ℓ the set of already bound labels was E_1, and we get the evidence [ℓ] # E_1. By the time we want to attach ℓ, an additional (existentially quantified) number of labels E_2 have been bound by the compilation of e. Consequently, we will need evidence [ℓ] # $(E_1 \cup E_2)$ to be able to safely attach ℓ. Proving this requires explicit reasoning about disjointness, combining the evidence returned by freshLabel with the evidence returned by the recursive invocation to compile.

These problems are symptomatic of the anti-compositional nature of typing label binding in bytecode. To some extent, this was to be expected: label names have to be globally well-bound and unique. Contextual bytecode typing enforces this as a whole-program property. The monadic compiler on the other is mostly manipulating *partial* bytecode programs, as it constructs its output piecewise. Hence, we are facing the question how we can *generalize* whole program typing to partial programs. The only way to do this is via *side conditions* that express how individual operations contribute to a proof that exceeds their local scope: the proof of the top-level conditions.

Instead of further pursuing this generalization of contextual typing, we propose a *co-contextual* reformulation of bytecode where labels are *nameless*. Using our nameless encoding we avoid the difficulties that we sketched above. The main reason for this is that a nameless representation rules out any sort of *accidental overlap* between labels. That is, unless a function gets passed a label explicitly, it has no means to get hold of that label. As a consequence, it is unnecessary for compile to provide evidence that it does not bind

the labels that we intend to attach later. It could not possibly do this, because we have not informed it about the existence of these labels. As we will see at the end of this section, our nameless representation makes it possible to assign strong types to the monadic operations on bytecode that intrinsically ensure label well-boundedness.

The means by which we obtain this nameless representation is via a *co-contextual* typing of bytecode with a *proof-relevant* notion of co-context merging. We introduce this idea first using typing rules, so as to make it easy to compare to the contextual typing rules in figure 5.3. We say that a typing rule is co-contextual[18,19] if instead of receiving a context, it produces a co-context. Co-contexts must pertain to the term at hand and contain no information that is irrelevant to the term. This is useful in dependently typed programming, which benefits from expressing invariants as *local knowledge*[20].

For bytecode, co-contexts are pairs $E \Updownarrow I$ of an *export* context E and *import* context I, which we call *interfaces* K. The roles of E and I are analogous to the E and I in the contextual rules. However, they are both *lists* of label types (rather than maps) because we have done away with names. Additionally, I is constrained to be the smallest possible in the same way as the linear context E which is already naturally in a co-contextual style. For example, nil has context $\epsilon = [] \Updownarrow []$. The resulting rules are summarized in figure 5.5, highlighting the differences with a shaded background.

[18] Erdweg et al. 2015. "A co-contextual formulation of type rules and its application to incremental type checking"

[19] Kuci et al. 2017. "A co-contextual type checker for Featherweight Java"

[20] McBride 2014. "How to keep your neighbours in order"

Figure 5.5:
Co-contextually-typed bytecode.

▼

Co-Pop
$$\frac{I = []}{I \vdash \mathsf{pop} : \langle a :: \psi \rightsquigarrow \psi \rangle}$$

Co-Push
$$\frac{k : a \qquad I = []}{I \vdash \mathsf{push}\ k : \langle \psi \rightsquigarrow a :: \psi \rangle}$$

Co-Goto
$$\frac{I = \psi_1}{I \vdash \mathsf{goto} : \langle \psi_1 \rightsquigarrow \psi_2 \rangle}$$

Co-Plain
$$\frac{I \vdash i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \qquad E = []}{E \Updownarrow I \vdash \mathsf{plain}\ i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle}$$

Co-Labeled
$$\frac{I \vdash i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \qquad E = \psi_1}{E \Updownarrow I \vdash \mathsf{labeled}\ i : \langle \psi_1 \rightsquigarrow \psi_2 \rangle}$$

T-Nil
$$\frac{K = \epsilon}{K \vdash \mathsf{nil} : \langle \psi \rightsquigarrow \psi \rangle}$$

Co-Cons
$$\frac{K_1 \vdash c : \langle \psi_1 \rightsquigarrow \psi_2 \rangle \qquad K_1 \bullet K_2 \simeq K \qquad K_2 \vdash b : \langle \psi_2 \rightsquigarrow \psi_3 \rangle}{K \vdash \mathsf{cons}\ c\ b : \langle \psi_1 \rightsquigarrow \psi_3 \rangle}$$

How does the nameless representation work? The goto and labeled operations no longer contain labels, so how does one know where to jump to? The key idea—inspired by McBride's co-de-Bruijn

representation[21]—is that type derivations (which in the intrinsically typed version will be part of the syntax) contain that information. In particular, the ternary relation $K_1 \bullet K_2 \simeq K$ for interface composition (or interface separation, depending on the perspective), is *proof relevant*. This means that proofs of $K_1 \bullet K_2 \simeq K$ are values that describe how reference occurrences appearing in I relate to binding occurrences in E.

[21] McBride 2018. "Everybody's got to be somewhere"

Why does this address the problems with the contextual typing from figure 5.3? As we already emphasized, by going nameless we no longer have accidental overlap between labels. This means that unlike the linear context separation relation $E_1 \sqcup E_2 \simeq E$ (appearing in the contextual typing rule T-Cons), the co-contextual counterpart $K_1 \bullet K_2 \simeq K$ (in Co-Cons) is not a proof *obligation*. Rather, it represents a *choice* how to relate the labels in K_1 with the labels in K_2. There is one trivial proof of $K_1 \bullet K_2 \simeq K$: the labels in K_1 and K_2 are independent. This is the proof that we choose when we append the output of two recursive calls of the compiler, for example, and which requires no additional evidence. In other words: unlike in the contextual formulation, avoiding accidental label binding requires *no* cooperation from the compiler writer. Apart from the trivial proof, one can construct non-trivial proofs of $K_1 \bullet K_2 \simeq K$ that relate references in K_1 with binders in K_2, and vice versa.

Another important aspect of interface composition $K_1 \bullet K_2 \simeq K$, is that binding and reference occurrences *cancel* each other out. This formalizes the idea that the reference occurrences are also obligations that are fulfilled by corresponding binding occurrences. All copies of the same reference occurrence are canceled out by a single binding occurrence. Because of this, labels that are used *and* defined in a fragment of bytecode, and are not used outside of this fragment, do *not appear in the exports of that fragment*. This effectively achieves scoping of labels, but without adding more structure to the bytecode that is not really present in the language. This *logical scoping* of labels allows us to prove the inclusion condition locally, rather than as a whole-program property.

The question now becomes how we write programs using nameless labels. As we will see in section 5.3, the actual definition of the proof-relevant interface composition relation $K_1 \bullet K_2 \simeq K$ is complicated. Certainly, if we have to manually work with the definition of that relation, then we fail to accomplish our goal of writing proof-

free compilers. This brings us to the remaining idea: abstracting over interfaces and their composition using an embedding of *proof-relevant separation logic*, and treating binding occurrences as *linear values* in a compiler.

To define the compiler, we need to define a type family Bytecode that internalizes the co-contextual typing rules in the nameless bytecode syntax. The definition of this type family can be found in section 5.7, but for now, we are content with only defining the syntax of intrinsically typed *labels*. We need two variations: Binder for binding occurrences, and Reference for reference occurrences:

```
data Binder    : StackTy → Intf → Set where
  binder    : Binder ψ    ([ ψ ] ⇅ [])

data Reference : StackTy → Intf → Set where
  reference : Reference ψ ([] ⇅ [ ψ ])
```

Agdaism: The type Intf is the representation of interfaces and has constructor _⇅_. It is defined precisely in section 5.3.

The stack type ψ types the occurrences: the binding occurrence must be attached to an instruction that expects a stack typed ψ, and the reference occurrence must be used by a jump instruction in a stack that has type ψ. The interface of the binding occurrence consists of the singleton export of ψ, whereas the reference occurrence consists of the singleton import of ψ.

We can now give the type of the touchstone of our nameless representation of labels:

Figure 5.6: A constant "fresh" pair of a binder and a reference to it.

▼

```
freshLabel : ∃⟨ (λ K₁ K₂ → Binder ψ K₁ × (K₁ • K₂ ≃ ε) × Reference ψ K₂) ⟩
```

That is, freshLabel is a *constant* pair of a binding and reference occurrence. It is constant, because there is no need to invent a unique name. The binding relation is established by the glue that sits in between: the composition of the interfaces $K_1 \bullet K_2 \simeq \epsilon$. The singleton export and import cancel each other out, so that the pair itself has the empty interface ϵ. This means that this compound object is internally well-bound: it does not import or export any labels from the surroundings. This naturally means that the occurrences are fresh with respect to any other label in the surroundings.

The final step is to realize that we can write this more succinctly if we abstract everywhere over interfaces and use the $*$ connective of separation logic[22]

```
freshLabel : (Binder ψ ∗ Reference ψ) ε
```

[22] As defined in section 4.4.

Separation logic provides a *high-level* language in which we can express the types and implementation of operations on the co-contextually typed bytecode at a suitable level of abstraction. From the perspective of the logic, label interfaces are a proof-relevant resource, just like linear typing contexts (section 4.5) and session contexts (section 4.6).

In section 5.3 we discuss how interface composition gives rise to a proof-relevant separation algebra. This yields a well-behaved model of separation logic on predicates over interfaces Intf (section 4.4). We use this high-level language as a logical framework for defining the intrinsically typed, co-contextual version of bytecode in section 5.7, and also to type all operations on bytecode and nameless labels. As a glimpse forward, we now give the signature of the typed compiler:

Figure 5.7: Signatures of the compiler and compilation operations that we define in section 5.8.

▼

```
Bytecode : StackTy → StackTy → Pred Intf

Compiler : StackTy → StackTy → Pred Intf → Pred Intf
Compiler ψ1 ψ2 P = Bytecode ψ1 ψ2 ∗ P

return  : ∀[ P ⇒ Compiler ψ ψ P ]
bind    : ∀[ (P —∗ Compiler ψ2 ψ3 Q) ⇒ (Compiler ψ1 ψ2 P —∗ Compiler ψ1 ψ3 Q) ]
tell    : ∀[ Bytecode ψ1 ψ2 ⇒ Compiler ψ1 ψ2 Emp ]
attach  : ∀[ Binder ψ1 ⇒ Compiler ψ1 ψ2 P —∗ Compiler ψ1 ψ2 P ]
compile : Exp a → ϵ[ Compiler ψ (a :: ψ) Emp ]
```

If you squint your eyes so that the separating conjunction $*$ becomes a normal product, the magic wand $—*$ becomes a normal function arrow, and Emp becomes $\top$, then the monad is "just" a parameterized[23] writer monad with the usual return, bind, and tell. The indexing with stack types ψ_1 and ψ_2 is used to type the bytecode output[24]. The fact that attach operation takes a *binding* occurrence of a label is made apparent in its type. The type of attach also expresses that the type of the label must match the entry type ϕ_1 of the output that it will be attached to.

The separation logic connectives help us abstract over interfaces and their compositions. This works uniformly for the low-level operations of compilation and for expression compilation, despite the fact that only expression compilation really preserves label well-boundness. We will show that the same logic is useful to define the typed syntax, and non-monadic functions over syntax.

[23] Atkey 2009. "Parameterised notions of computation"

[24] McKinna et al. 2006. "A type-correct, stack-safe, provably correct, expression compiler"

5.3 *A Model of Nameless Co-contextual Binding*

We present a model of nameless co-contextual global binding. Its elements K are *binding interfaces*, which abstractly describe the label binders and label references of a syntax fragment. Interfaces are composed using a ternary proof-relevant relation $K_1 \bullet K_2 \simeq K$. We first present the model using a handful of examples that demonstrate its key features and then define it formally. Finally, we explain why proof relevance is essential, by considering the translation from labels to absolute addresses.

We define *label interfaces* K : Intf as pairs of label typing contexts LabelCtx:

```
StackTy  = List Ty
LabelCtx = List StackTy

record Intf : Set where
  constructor _⇅_
  field
    exp  : LabelCtx
    imp  : LabelCtx

ϵ : Intf
ϵ = [] ⇅ []
```

Agdaism: Intf is a record with constructor $(E \Uparrow I)$.

The projections exp and imp describe the label binders (the *exports*) and label references/jumps (the *imports*), respectively. We write ϵ for the empty interface.

The fact that we use a nameless representation of label binding is visible in two ways. First, label contexts LabelCtx are mere lists.[25] Second, the relation $K_1 \bullet K_2 \simeq K$ for composition of interfaces K_1 and K_2 into K is proof relevant—i.e., proofs of $K_1 \bullet K_2 \simeq K$ are values that choose between options of relating the labels in K_1 to those in K_2. Before we formally define this relation, we illustrate its key features through a number of examples.

[25] In some sense label contexts are really multisets or bags, because everything must be stable under permutation of the lists. The proof term of composition does, however, depend on the order of elements. More details are given in section 5.4

The simplest way to compose interfaces—which is applicable for any choice of operand interfaces—is to take the disjoint union of the imports and exports of the interface:

$$(E_1 \Uparrow I_1) \bullet (E_2 \Uparrow I_2) \simeq ((E_1 \mathbin{++} E_2) \Uparrow (I_1 \mathbin{++} I_2)) \tag{1}$$

In this case there is no interaction between the binding of the parts, and their label use is completely disjoint. A concrete instance of the

above composition would be to compose two interfaces that both have an import (i.e., a label reference) of type ψ:

$$([] \Downarrow [\ \psi\ ]) \bullet ([] \Downarrow [\ \psi\ ]) \simeq ([] \Downarrow [\ \psi\ ,\ \psi\ ]) \tag{2}$$

The interfaces $([] \Downarrow [\ \psi\ ])$ could be assigned to two bytecode sequences that both contain a single jump instruction with target stack type ψ. By taking the disjoint union, we express that these jump instructions refer to disjoint label binders. If we wish to express that these jump instructions refer to the same binder, then the composition relation can contract them. For example:

$$([] \Downarrow [\ \psi\ ]) \bullet ([] \Downarrow [\ \psi\ ]) \simeq ([] \Downarrow [\ \psi\ ]) \tag{3}$$

Equation (1) and equation (3) show that the relation $K_1 \bullet K_2 \simeq K$ is not functional—there is a choice whether to contract imports in K_1 or K_2, or not. Apart from not being functional, the relation $K_1 \bullet K_2 \simeq K$ is also proof relevant. The different proofs the relation are values that represent the different choices of composing K_1 and K_2. For example, consider a composition of two interfaces where the left has one import of type ψ, and the right and composite interfaces have two imports of type ψ:

$$([] \Downarrow [\ \psi\ ]) \bullet ([] \Downarrow [\ \psi\ ,\ \psi\ ]) \simeq ([] \Downarrow [\ \psi\ ,\ \psi\ ]) \tag{4}$$

The interface $([] \Downarrow [\ \psi\ ])$ on the left could be assigned to bytecode that contains a single jump instruction with target stack type ψ, while the interface $([] \Downarrow [\ \psi\ ,\ \psi\ ])$ on the right could be assigned to bytecode that contains two jump instructions with target stack type ψ. We can now contract the import on the left in two ways—do we wish to contract it with the first or the second import on the right? Since we consider the composition relation to be proof relevant, there are two proofs of equation (4) corresponding to this choice.

We have seen that the interface composition relation can be used to contract imports in K_1 and K_2, which corresponds to expressing different jump instructions refer to the same target binder. Contraction of exports is impossible, as it would not make sense to express that two binders should become the same target. Thus, for example, we do *not* have:

$$([\ \psi\ ] \Downarrow []) \bullet ([\ \psi\ ] \Downarrow []) \simeq ([\ \psi\ ] \Downarrow []) \tag{5}$$

Given the sub-structural nature of composition, we conclude that exports are treated essentially as *linear*, and imports as *relevant*. This agrees with the co-contextual typing style.

We now turn to binding an import to an export. Using composition $K_1 \bullet K_2 \simeq K$, we may bind an import from K_1 to an export from K_2 of the same type, and vice versa. For example, we can bind an import typed ϕ from the left, using an export typed ϕ from the right:

$$([] \Downarrow [\ \phi\ ,\ \psi\ ]) \bullet ([\ \phi\ ] \Downarrow []) \simeq ([\ \phi\ ] \Downarrow [\ \psi\ ]) \tag{6}$$

The interface $([] \Downarrow [\ \phi\ ,\ \psi\ ])$ on the left could be assigned to bytecode that contains two jump instructions to label references of types ϕ and ψ, while the interface $([\ \phi\ ] \Downarrow [])$ on the right could be assigned to bytecode that contains no (unbound) jump instructions, but a binding occurrence of type ϕ. The composition expresses that the reference occurrence of the label of type ϕ on the left is bound to the binding occurrence on the right. Because the import of ϕ is fulfilled—i.e., the jump instruction to ϕ has been bound—ϕ does not reappear in the imports of the composite.

When binding an import using an export, there is a choice whether we want to use the export again or not. In equation (6), the export ϕ remains available in the composition, which means that another jump instruction can target it as well. Instead, we could decide to *hide* the export ϕ:

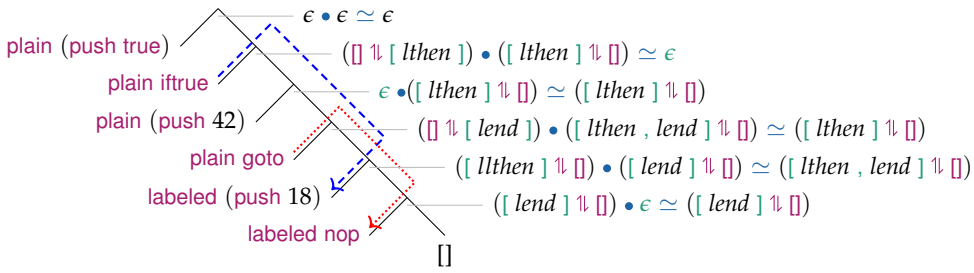
$$([] \Downarrow [\ \phi\ ,\ \psi\ ]) \bullet ([\ \phi\ ] \Downarrow []) \simeq ([] \Downarrow [\ \psi\ ]) \tag{7}$$

Hiding an export is only allowed if it has been bound by an import. In other words, if we choose *not* to bind an import using an export, then we cannot hide the export. This ensures that the rules are truly co-contextual by keeping the label contexts tight (and also rules out unused label declarations). This ensures that the model has no mysterious compositions with the unit ϵ, like:

$$\epsilon \bullet ([\ \psi\ ] \Downarrow []) \simeq \epsilon - \textit{counterexample} \tag{8}$$

We make the properties of identity compositions more precise in the next section.

The hiding gives rise to *logical scopes*, which are not present in the syntactical structure, but are present in the proof-relevant binding model. Figure 5.8 shows how the various features of the composition relation can be used to type the bytecode corresponding to the conditional expression.



The tree represents the list-like bytecode abstract syntax, where every node is annotated with the interface composition. Since we use a nameless representation, the only way to tell which label goes where (visualized using the blue and red arrow) is by examining the proofs that witness the composition. The labels are only in scope in the parts of the tree that the dotted lines traverses. Most notable, they can be in scope in the *spine* of the tree, without being in scope of some of the instructions that are attached to it.

The figure also shows how labels are hidden from instructions higher in the sequence using export hiding: when we bind a labeled instruction to a goto (respectively, iftrue), the corresponding export of type int :: ψ (respectively, ψ) is hidden, because there are no further uses elsewhere.

▲

Figure 5.8: Interface compositions in the result of compilation of if true then 42 else 18. For every composition $K_1 \bullet K_2 \simeq K$, the interfaces K_1 and K_2 describe the imports and exports of the head and tail of that node respectively, whereas K does the same for the entire node. Hence, if a node is labeled with a composition $K_1 \bullet K_2 \simeq K$, then the node above it must have a composition of the form $(\ldots) \bullet K \simeq (\ldots)$. The dashed blue path and dotted red path represent the effective binding of *lthen* and *lend* respectively.

5.4 *Interface Composition, Formally*

We now present the formal definition of interface composition $K_1 \bullet K_2 \simeq K$. The formal definition must clarify (1) which imports from K_1 are bound by exports of K_2 and vice versa, and (2) which imports and exports from K_1 and K_2 are exposed by the composite K. We take care to specify this in a way that ensures that each bound import resolves *unambiguously* to a single export. In particular, this means that each import that is bound to an export *cannot* reappear in the composite imports (see equation (6)). Conversely, each import that is not bound *must* appear in the import list of the composite K. In addition, exports are treated *relevantly*: if K_1 (respectively, K_2) exports a binder that is not used to bind an import of K_2 (respectively, K_1), then it *must* reappear in the exports of the composite K. But, if exports in either K_1 (respectively, K_2) are bound by an input in K_2 (respectively, K_1) then they *can* be hidden in K.

To define interface composition, we first define two PRSAs for disjoint label-context separation and overlapping label-context sepa-

ration. The former will model linear use of label binders, and the latter will model relevant use of label references. Disjoint- and overlapping context separation are both instances of a generalized PRSA for the separation of bags of elements of some resource $\mathcal{R}$. A natural definition of this generalized notion is to take our generalized PRSA for list separation (figure 4.7) and superimpose proof relevant list permutations $xs \leftrightsquigarrow ys$:

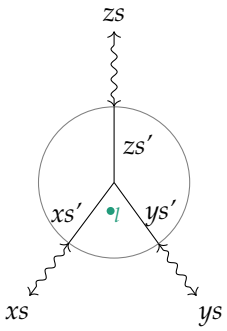
Figure 5.9: Defining the proof-relevant separation of bags by superimposing permutations onto proof-relevant list separation.

▼

```
record BagSplit (xs ys zs : List ℛ) : Set where
  field
    {xs' ys' zs'} : List ℛ

    – superimposed permutations
    ρx : xs' ⇜⇝ xs
    ρy : ys' ⇜⇝ ys
    ρz : zs' ⇜⇝ zs

    – underlying list separation
    sep : xs' •ₗ ys' ≃ zs'
```



It is straightforward to prove that this ternary relation is again a PRSA for any list element division relation that is a proof-relevant partial semigroup respecting list permutations $\leftrightsquigarrow$.

We obtain two PRSAs for disjoint (_⊔_≃_) and overlapping (_∪_≃_) label-context separation by instantiating the bag separation relation in two ways. For the former, we take list element division to be the empty relation. For the latter, we pick a relation that duplicates an element[26]. We effectively get:

```
_⊔_≃_ : LabelCtx → LabelCtx → LabelCtx → Set
_∪_≃_ : LabelCtx → LabelCtx → LabelCtx → Set
```

Both are PRSAs respecting list permutations.

[26] That is, the following ternary relation:

```
data Dup : ℛ → ℛ → ℛ → Set where
  dup : Dup r r r
```

Figure 5.10: Interface composition relation.

▼

Interface composition is now defined as follows:

```
data Binds : (E I E′ I′ : LabelCtx) → Set where
  binds :  (I′ ⊔ Δ ≃ I)
        → (E′ ∪ Δ ≃ E)
        → Binds E I E′ I′
```

```
data _•_≃_ : Intf → Intf → Intf → Set where
  comp :  Binds E₁ I₂ E₁′ I₂′
       → Binds E₂ I₁ E₂′ I₁′
       → (E₁′ ⊔ E₂′ ≃ E) → (I₁′ ∪ I₂′ ≃ I)
       → ((E₁ ⇅ I₁) • (E₂ ⇅ I₂) ≃ (E ⇅ I))
```

The composition relation expresses that the composite $(E \Downarrow I)$ of $(E_1 \Downarrow I_1)$ and $(E_2 \Downarrow I_2)$ is obtained by (1) binding some of the imports I_2 to exports E_1, resulting in "leftovers" I_2' and E_1', and symmetrically binding certain I_1 to E_2, resulting in "leftovers" I_1' and E_2', (2) adding up the leftover imports and exports to obtain E and I. Exports are combined *without* overlap (i.e., using $\sqcup$), while imports are combined *with* overlap (i.e., using $\cup$), implementing import contraction. The intuition behind the asymmetry between the use of disjoint separation for imports and separation with overlap for exports comes from the fact that labels should be bound only once, whereas they can be referenced multiple times.

In order to express which imports are bound to which exports, we define the auxiliary relation Binds $E\ I\ E'\ I'$. This relation specifies that E' and I' consist of the "leftover" exports and imports after *binding* some of the imports I to the exports E. It disjointly separates the imports I into those to be bound Δ, and those that are leftover I'. As shown in the definition of interface composition, the leftover imports will have to reappear in the imports of the composite interface. The bound imports Δ are also subtracted from the exports E to get the leftover exports E'.

There can be overlap between E and E', giving the option to re-export used exports from the composite, as shown in equation (6). Or, conversely, we only have the choice to hide an export from an interface if it has been bound by an import, thus precluding the composition in equation (8).

Since our binding model uses nameless label binders and references, it is essential that the interface composition relation is proof relevant. That is, labels are given a meaning not by the mere fact that there *exist* witnesses for interface composition in all nodes of the typing derivation, but by the *specific* witnesses that have been used to construct the typing derivation. Although cryptic in its form, the composition relation really lays out the binding paths visualized in figure 5.8.

The proof relevance becomes most apparent in the last pass of our compiler backend (section 5.9), where our nameless representation is translated into a representation with jumps to absolute addresses. This transformation works by computing an environment that assigns an address to every label in the imported label context I. This environment is split and recombined along the witnesses of the

interface composition in each node of the typing derivation. This way, jumps follow the path laid out by the witnesses of the interface composition relation.

5.5 *Interface Composition as a PRSA*

The technical material of this section was not earlier described in a paper, but was part of the submitted code artifact of Rouvoet et al. 2021.

This is a highly technical section describing a proof structure. The details of the proof are not required to be understood for the remainder of this chapter.

To make use of the logical connectives of the proof relevant separation logic that we developed in section 4.4, we need to prove that interface composition is a PRSA. In this section we prove this *generically*. That is, we abstract interfaces, the interface composition relation $\_\bullet\_\simeq\_$ and the auxiliary relation Binds (figure 5.10) from LabelCtx to an arbitrary carrier $\mathcal{R}$. We also abstract over the proof relevant separation algebras $\_\sqcup\_\simeq\_$ and $\_\cup\_\simeq\_$. We then prove the generalized interface composition to be a PRSA by requiring the $\sqcup$ and $\cup$ ternary relations to obey generalized *cross-split*[27] properties. This provides some insight into the type of instances that we can expect to exist.

[27] Dockins et al. 2009. "A fresh look at separation algebras and share accounting"

We have to prove the identity laws ($\bullet$-id^l and $\bullet$-id^{-l}), commutativity ($\bullet$-comm), and associativity ($\bullet$-assoc). Of these laws, associativity is the challenging one. We will focus on that law in this section. The proof of associativity only comprises 15 lines of Agda code, but understanding it requires some insight which we will clarify subsequently using a graphical depiction of the proof.

The instance we have to prove is:

$$
\begin{aligned}
\bullet\text{-assoc} : \ & (\sigma_1 : (A \Downarrow a) \quad \bullet\ (B \Downarrow b) \simeq (AB \Downarrow ab)) \\
& \to (\sigma_2 : (AB \Downarrow ab) \bullet (C \Downarrow c) \simeq (ABC \Downarrow abc)) \\
& \to \exists\langle\ (\lambda\ bc\ BC \to (A \Downarrow a\ ) \bullet (BC \Downarrow bc\ ) \simeq (ABC \Downarrow abc\ ) \\
& \qquad\qquad \times (B \Downarrow b\ ) \bullet (C\ \ \Downarrow c\ \ ) \simeq (BC\ \ \Downarrow bc\ \ ))\ \rangle
\end{aligned}
$$

Where we adopt the convention to write exports in uppercase, and imports in lowercase. Composites (e.g., AB) are given names derived from their parts (i.e., A and B).

Each of the two premises can be deconstructed into six compositions of elements of $\mathcal{R}$, yielding twelve compositions in total. These can be organized into six compositions on exports, and six compositions on imports. This yields two composition diagrams, which can be depicted as follows:

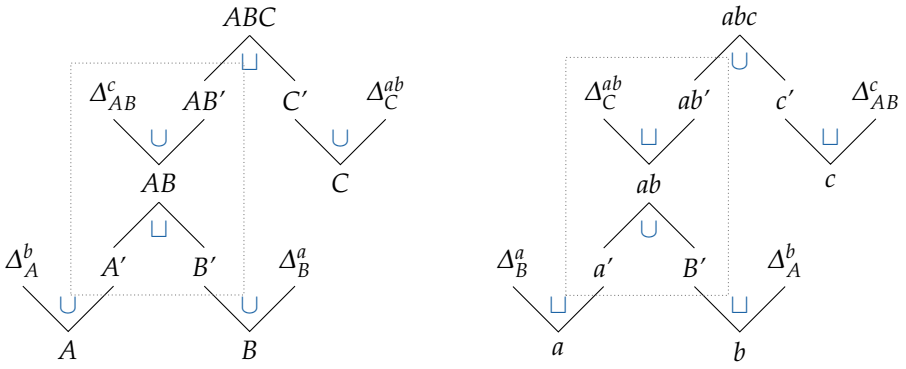
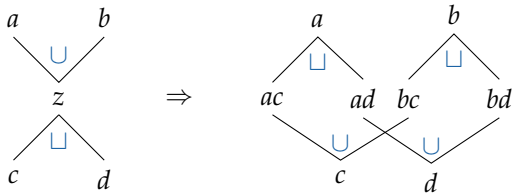


Figure 5.11: Initial interface composition trees. Capitals denote exports and lowercase denotes imports. The notation Δ^b_A says "the exports from A being bound to imports from b". The boxes mark the section that we will rewrite using cross-split. ▲

In both diagrams, the top three compositions come from premise σ_1 and the bottom three compositions come from premise σ_2. We write X' (respectively x') for the *leftovers* of X (respectively x). The subtracted amount we write Δ^z_Y, with the understanding that some amount of exports Y are being used to bind imports z. Note that the export and import diagrams *exchange* these bound elements. Also note that the $\cup$ and $\sqcup$ relations are opposite in the two diagrams.

We remark that both diagrams are such that we can not apply associativity of either the $\cup$ or $\sqcup$ relation anywhere. Note also that the diagram contains two compositions of AB (respectively ab). To make use of associativity of the underlying relations, we need to push the $\sqcup$-separation of AB through the $\cup$-separation. To this end, we will require a generalized cross-split[28] property between the relations for $\sqcup$ and $\cup$:[29]

$$
\begin{aligned}
&\text{cross-split} : \\
&\quad (a \cup b \simeq z) \to (c \sqcup d \simeq z) \to \\
&\quad \exists\langle\, (\lambda\ ac\ ad\ bc\ bd \to (ac \sqcup ad \simeq a) \times (bc \sqcup bd \simeq b) \\
&\qquad\qquad \times (ac \cup bc \simeq c) \times (ad \cup bd \simeq d))\ \rangle
\end{aligned}
$$



The cross-split property of Dockins et al. (2009) essentially says that if one can cut a cake in two halves in two ways, then surely one can also cut it in four quarters, such that the quarters are made

[28] Dockins et al. 2009. "A fresh look at separation algebras and share accounting"

[29] The original property is generalized to PRSAs and to two notions of separation, rather than one.

up from the intersections of the four halves. We can depict our generalized cross-split property pictorially as follows:
Where the four quarters *ac–bd* and the witnesses of the four compositions are existentially quantified. We apply this law to the marked sections in the initial export and import diagrams, and obtain the following diagrams:

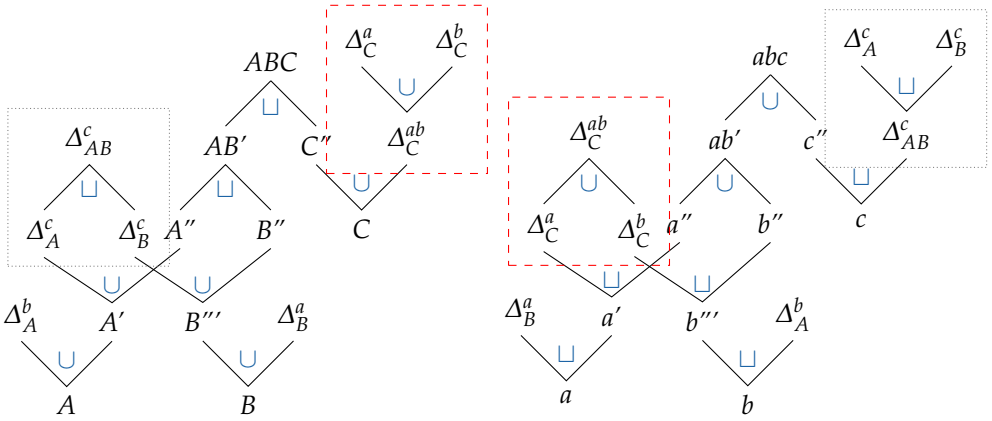


Figure 5.12: Step 1: after rewriting figure 5.11 using cross-split. The boxed sections are transported between diagrams to accommodate re-association of C and c.

Now we want to re-associate the compositions of A, B, and C (respectively a, b, and c). For A and B (respectively a and b) this is readily possible. For C (respectively c) we need a decomposition of Δ^c_{AB} (respectively Δ^{ab}_C). We transport these decompositions from the opposite diagram—we marked the relevant decompositions in the diagrams above.

After applying the re-associations, we obtain the diagram in figure 5.13. From there, we want to use the inverse of the cross-split property to obtain the re-association of the initial diagrams. To be able to do this we again transported some compositions from the export to the import diagram and vice versa. More precisely, we transport the composition of Δ^a_{BC} from the import diagram to the export diagram, and we transport the composition of Δ^{bc}_A from the export diagram to the import diagram. The transported composition pairs are marked in figure 5.13.

▲

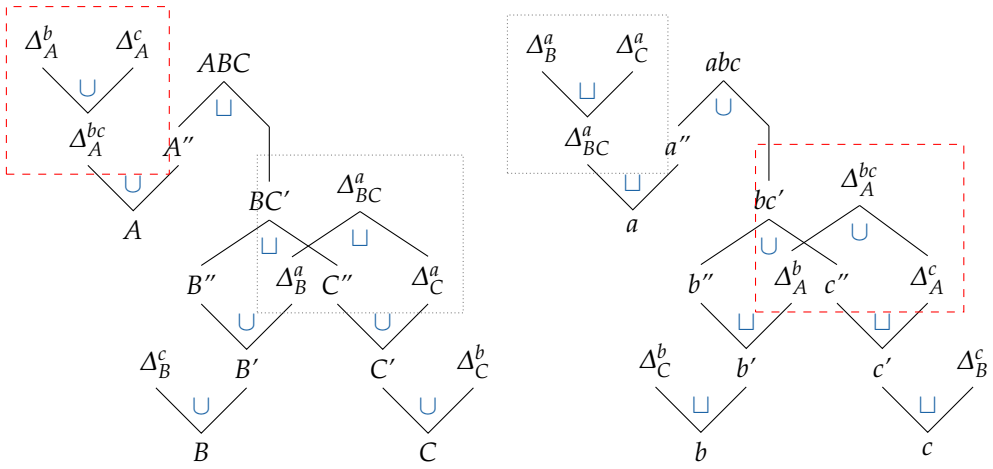


Figure 5.13: Step 2: Re-associate ABC, A, B, and C (respectively abc, a, b, and c). Boxed sections are again transported between diagrams.

▲

The last step is then to apply the inverse of the cross-split property, which is the second assumption about the relations $\_\sqcup\_\simeq\_$ and $\_\cup\_\simeq\_$. The final diagram is shown in figure 5.14 below.

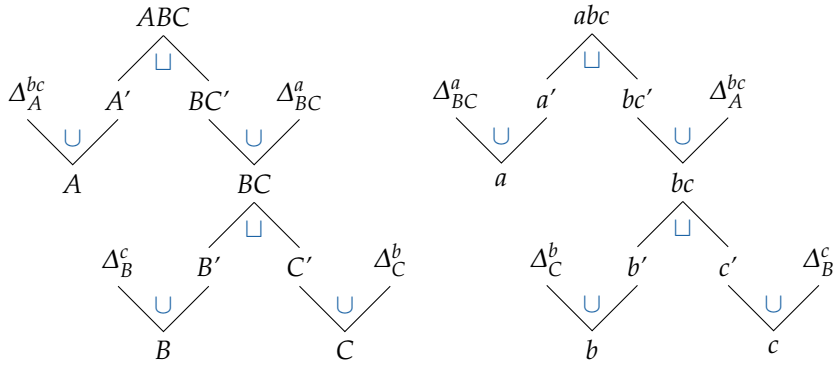


Figure 5.14: Step 3: apply the inverse of cross-split.

▲

We have now shown that the ternary relation $\_\bullet\_\simeq\_$ on interfaces is associative. This is the main result of proving it to be a proof relevant separation algebra. Commutativity is straightforward, and so are the lemmas that say that it respects the natural notion of equivalence in all three positions of the relation. Proving the identity laws requires two more assumptions. In particular, we need to show that both relations on the carrier $\mathcal{R}$ are *positive*. That is, the empty element in $\mathcal{R}$ can only be decomposed into two empty elements:

$$\sqcup\text{-isPositive} : E_1 \sqcup E_2 \simeq \epsilon \to (E_1 , E_2) \equiv (\epsilon , \epsilon)$$
$$\cup\text{-isPositive} : I_1 \cup I_2 \simeq \epsilon \to (I_1 , I_2) \equiv (\epsilon , \epsilon)$$

The disjoint separation of label contexts and the separation of label contexts with overlap satisfy all the assumptions.[30] Noteworthy is that the cross-split assumptions do not hold for maps of identifiers to types. The nameless representation of labels is key to enabling the *hiding* of labels. When one tries to prove the associativity laws for maps, the proof gets stuck because the re-association may extend the scope of a hidden label over syntax that defined the same name.

[30] Of the necessary assumptions, the cross-split property and its inverse are the hardest (and lengthiest) to prove. The cross-split property for generalized bag separation (approx. 30 lines of Agda) follows from the fact that it holds for the underlying list separation, provided that it holds for the element division relation. The inverse of cross-split only holds for bag separations (approx. 45 lines of Agda).

5.6 *Separation Logic for Interfaces*

To specify intrinsically-typed bytecode syntax (section 5.7) and to implement our compiler (section 5.8) we instantiate our separation logic from section 4.4 with three different PRSAs: (1) disjoint context separation _⊔_≃_ to abstract over exports (i.e., label binders), (2) context separation with overlap _∪_≃_ to abstract over imports (i.e., label references), and (3) interface composition _•_≃_ to abstract over interfaces. To convert between these three separation logic instances, we define modalities Export and Import for lifting objects from the logics of label contexts to the logic of interfaces:[31]

[31] These modalities and their laws also hold up for the more abstract PRSA that we used in the previous section.

```
data Export (P : Pred LabelCtx) : Pred Intf where
  exports : P E → Export P (E ⇂ [])

data Import (P : Pred LabelCtx) : Pred Intf where
  imports : P I → Import P ([] ⇂ I)
```

Both modalities are monotone and commute with the separating conjunction:

```
mapExport : ∀[ P ⇒ Q ] → ∀[ Export P ⇒ Export Q ]
mapImport : ∀[ P ⇒ Q ] → ∀[ Import P ⇒ Import Q ]
zipExport  : ∀[ (Export P) ∗ (Export Q) ⇒ Export (P ∗ Q) ]
zipImport  : ∀[ (Import P) ∗ (Import Q) ⇒ Import (P ∗ Q) ]
```

Note that the separating conjunction ∗ in Export (P ∗ Q) is using disjoint context separation _⊔_≃_, whereas the ∗ in Import (P ∗ Q) is using context separation with overlap _∪_≃_, and the ∗ at the top-level is using interface composition _•_≃_. The fact that these modalities convert from one logic to another means that they are *relative*[32].

[32] Altenkirch et al. 2015. "Monads need not be endofunctors"

The intuition for these dual modalities is that they generalize the dual roles of labels as either imports (i.e., binders) or exports (i.e.,

references). We can thus recover the interface predicates Binder and Reference from section 5.2 as follows:

```
Binder ψ     = Export (Own [ ψ ])
Reference ψ = Import (Own [ ψ ])
```

From now on, we will use the right-hand side of these definitions directly, highlighting the role of the modalities in the bytecode syntax and the monadic operations.

In section 5.3 we showed how the exports and imports of interfaces interact. This interaction is internalized in separation logic over interfaces by the following operation:

```
freshLabels : ϵ[ Export (Own E) ∗ Import (Own E) ]
```

This operation generalizes the freshLabel operation we have seen in section 5.2 from Binder and Reference to the Export and Import modalities, thus allowing to generate multiple labels. Like matter and anti-matter, the operation freshLabels shows that labels can spontaneously come into existence in pairs of binding and reference occurrences. Binding occurrences are represented as ownership of a linear resource $E \Uparrow []$, whereas reference occurrences are represented as ownership of a relevant (i.e., duplicable) resources $[] \Uparrow E$. The definition of freshLabels relies on the fact that these resources are dual, and thus cancel each other out: $(E \Uparrow []) \bullet ([] \Uparrow E) \simeq \epsilon$.

5.7 *Intrinsically-Typed Nameless Co-Contextual Bytecode*

Using the separation logic as a logical framework, we now define intrinsically-typed bytecode. We adopt the perspective that well-typed terms are separation logic predicates on interfaces, thus abstracting over co-contexts and their compositions.

The typed bytecode language that we define is a subset of JVM bytecode[33]. Unlike JVM bytecode, we use labels as jump targets.[34] Labels can be translated away later in the compiler pipeline, after the bytecode is optimized (section 5.9). Compared to the simple bytecode languages from section 5.2, our JVM subset contains some additional features: local variables, and multiple labels for a single instruction. The latter simplifies compilation, but does not change the verification problem that we described in section 5.2.

[33] Lindholm et al. 2020. "The Java Virtual Machine specification: Java SE 14 edition"

[34] JVM bytecode uses instruction indices that are local to the class file.

The intrinsically-typed bytecode syntax internalizes the co-contextual typing judgments shown in figure 5.5. We first define plain instructions Instr $\Gamma\ \psi_1\ \psi_2$:

```
VarCtx = List Ty

data Instr (Γ : VarCtx) : (ψ₁ ψ₂ : StackTy) → Pred LabelCtx where
  nop   :                 ϵ[ Instr Γ ψ ψ ]
  pop   :                 ϵ[ Instr Γ (a :: ψ) ψ ]
  swap  :                 ϵ[ Instr Γ (a :: b :: ψ) (b :: a :: ψ) ]
  iadd  :                 ϵ[ Instr Γ (int :: int :: ψ) (int :: ψ) ]
  load  : a ∈ Γ →         ϵ[ Instr Γ ψ (a :: ψ) ]
  store : a ∈ Γ →         ϵ[ Instr Γ (a :: ψ) ψ ]
  goto  :                 ∀[ Own [ ψ₁ ] ⇒ Instr Γ ψ₁ ψ₂ ]
  iftrue :                ∀[ Own [ ψ ]  ⇒ Instr Γ (boolean :: ψ) ψ ]
```

Variable binding is modeled contextually using a local variable context Γ. As is usual, variables are modeled using proof-relevant membership of the context $a \in \Gamma$, pointing out a *specific* element in Γ. The indices ψ_1 and ψ_2 again denote the pre and post stack typing. The values of these indices for the various instructions are identical to the stack types in the extrinsic co-contextual rules. As before, plain instructions only import labels (i.e., only contain label references), and are thus typed as predicates over a LabelCtx. The jump instructions (goto and iftrue) are the only instructions that have imports, and use the predicate Own [ψ] to express that.

In contrast to instructions, code points Code $\Gamma\ \psi_1\ \psi_2$ are predicates on interfaces Intf, because they import and export labels (i.e., contain label references and binders):

Figure 5.15: Typed code points. We choose to distinguish between unlabeled and labeled instructions. Another candidate representation of code-points is a separating conjunction of a *possibly empty* labeling and an instruction. This works, but requires the application of the identity lemmas to reason away the empty labeling in functions operating on bytecode that happens to be plain. Making this common case perspicuous avoids this proof burden.

▼

```
Labeling ψ = Plus (Own [ ψ ])
data Code (Γ : VarCtx) : (ψ₁ ψ₂ : StackTy) → Pred Intf where
  labeled : ∀[ Export (Labeling ψ₁) ∗ Import (Instr Γ ψ₁ ψ₂) ⇒ Code Γ ψ₁ ψ₂ ]
  plain   : ∀[                       Import (Instr Γ ψ₁ ψ₂) ⇒ Code Γ ψ₁ ψ₂ ]
```

The constructors labeled and plain use the Import-modality to lift plain instructions into the logic of predicates on interfaces. The constructor labeled also attaches one or more binders to an instruction using the Export modality. For this it uses the type Labeling ψ, which represents one or more labels of the same type ψ, and is defined using a version of the Kleene plus type Plus in separation logic:

```
data Star (P : Pred A) : Pred A where
  nil  : ϵ[ Star P ]
  cons : ∀[ P ∗ Star P ⇒ Star P ]

Plus : Pred A → Pred A
Plus P = P ∗ Star P
```

Finally, bytecode `Bytecode Γ ψ1 ψ2` is represented as the indexed reflexive-transitive closure (or Kleene star) `IStar` of code points:

```
data IStar (R : I → I → Pred A) : (I → I → Pred A) where
  inil  : ϵ[ IStar R i i ]
  icons : ∀[ R i1 i2 ∗ IStar R i2 i3 ⇒ IStar R i1 i3 ]

Bytecode Γ ψ1 ψ2 = IStar (Code Γ) ψ1 ψ2
```

IT IS NOTEWORTHY that the data structures `Star`, `Plus` and `IStar` are defined generically over arbitrary PRSAs. This is a strength of the approach we have taken. By casting the invariants of label binding in the framework of proof-relevant separation logic, we have gained reuse of common data structures and operations that we otherwise would have had to tailor to a specific domain.

5.8 *Compiling with Labels*

Having defined our bytecode language in the previous section, we now describe an intrinsically-typed compiler that translates programs in an imperative source language with structured control into bytecode. We first define the well-typed syntax of the source language IMP. We then define a monad `Compiler`, which extends the interface of a writer monad with operations for generating bytecode, and is used to present our compiler.

THE INTRINSICALLY-TYPED SYNTAX of the imperative language IMP (see, for example, Nipkow et al.[35] for a plain definition) is depicted in Figure 5.16. Like many imperative languages, it distinguishes expressions `Exp Γ a` and statements `Stmt Γ` (also known as *commands*). Expressions are pure and result in a value of type `a`, whereas statements are side-effecting but have no result.

[35] Nipkow et al. 2014. "IMP: A Simple Imperative Language"

```
data BinOp : (a b c : Ty) → Set where
  add sub mul div xor : BinOp int int int
  eq ne lt ge gt le   : BinOp int int boolean

data Exp (Γ : VarCtx) : Ty → Set where
  num         : ℤ →                Exp Γ int
  bool        : Bool →             Exp Γ boolean
  var         : a ∈ Γ →            Exp Γ a
  bop         : BinOp a b c → Exp Γ a → Exp Γ b → Exp Γ c
  if_then_else :                   Exp Γ boolean → Exp Γ a → Exp Γ a → Exp Γ a

data Stmt (Γ : VarCtx) : Set where
  assign      : a ∈ Γ →            Exp Γ a → Stmt Γ
  if_then_else :                   Exp Γ boolean → Stmt Γ → Stmt Γ → Stmt Γ
  while       :                    Exp Γ boolean → Stmt Γ → Stmt Γ
  block       :                    List (Stmt Γ) → Stmt Γ
```

▲

Figure 5.16: Intrinsically-typed syntax of the imperative language IMP.

Expressions are constants (num, and bool), reads from local variables (var), binary operations for arithmetic and comparison (bop), or conditionals (if_then_else). Statements are assignments to local variables (assign), conditions (if_then_else, an overloaded constructor), while loops (while), or sequenced statements (block).

The language IMP misses a statement for local variable declarations. The compiler backend that we present in section 5.9 will start with an extended source language IMP$^+$ that includes local variable declarations. The first pass translates IMP$^+$ to IMP by hoisting local variables declarations. This separate pass simplifies the compilation to bytecode as it allows us to use the same context Γ for the source Exp Γ a and target Bytecode Γ ψ_1 ψ_2 of the compiler.[36]

In IMP$^+$, local variable declarations come with initializers, which ensures that variables are initialized before use. This is, however, not witnessed by the typing of IMP, where the first assignment is separated from the hoisted declaration. Intrinsically capturing the effect analysis for variable initialization is an interesting direction for future work, orthogonal from the verification problem with labels that we address in this chapter.

[36] For simplicity we use the same set of types for the IMP and bytecode language. The mechanization uses different sets of types for both languages. It shows how one can translate almost transparently between them using Agda's mechanism for type class search and rewriting (Cockx 2020).

Like the untyped compiler in figure 5.1, we use a writer monad to collect the output generated by the compiler. Compared to the ordinary writer monad, there are two twists. First, our writer monad is written in the category of separation logic predicates instead of

the category of plain types, so that the handling of label binding is encapsulated. We already saw an example of such monads in section 4.5 and section 4.6). Second, our writer monad is parameterized [37] so as to keep track of the pre and post stack types of the generated bytecode.

[37] Atkey 2009. "Parameterised notions of computation"

Before showing how our writer monad is used to generate bytecode (section 5.8), we present it in its general form. That is, we define it abstractly for a separation logic Pred A over any given PRSA A, and any given type $W : I \to I \to \mathsf{Pred}\ A$ indexed by I that represents the generated output. The writer monad is then defined as follows:

$$\mathsf{Writer} : (I \to I \to \mathsf{Pred}\ A) \to I \to I \to \mathsf{Pred}\ A \to \mathsf{Pred}\ A$$
$$\mathsf{Writer}\ W\ i_1\ i_2\ P = W\ i_1\ i_2 * P$$

This definition resembles the usual definition of a writer monad. However, since the monad is defined in terms of separation logic, we use the separating conjunction $*$ instead of an ordinary product. The indices i_1 and i_2 of the monad are taken to be the indices of the output W.

To define the monadic operations of our writer monad, the type W for the output must be an *indexed monoid*. That means, we need to have operations mempty for the empty output, and mappend for appending output:

$$\mathsf{mempty} : \epsilon[\ W\ i\ i\ ]$$
$$\mathsf{mappend} : \forall[\ W\ i_1\ i_2 \Rightarrow W\ i_2\ i_3 \mathrel{-\!\!*} W\ i_1\ i_3\ ]$$

These operations generalize the ordinary indexed monoid operations, but instead of using functions, we use the separation logic connectives.

Apart from return and bind, our writer monad has two additional monadic operations: tell for generating output, and censor for transforming the output of a given writer computation:

Figure 5.17: The writer monad interface.

▼

$$\mathsf{return} : \forall[\ P \Rightarrow \mathsf{Writer}\ W\ i_1\ i_1\ P\ ]$$
$$\mathsf{bind} : \forall[\ (P \mathrel{-\!\!*} \mathsf{Writer}\ W\ i_2\ i_3\ Q) \Rightarrow (\mathsf{Writer}\ W\ i_1\ i_2\ P \mathrel{-\!\!*} \mathsf{Writer}\ W\ i_1\ i_3\ Q)\ ]$$
$$\mathsf{tell} : \forall[\ W\ i_1\ i_2 \Rightarrow \mathsf{Writer}\ W\ i_1\ i_2\ \mathsf{Emp}\ ]$$
$$\mathsf{censor} : \forall[\ (W\ i_1\ i_2 \mathrel{-\!\!*} W\ i_3\ i_4) \Rightarrow \mathsf{Writer}\ W\ i_1\ i_2\ P \mathrel{-\!\!*} \mathsf{Writer}\ W\ i_3\ i_4\ P\ ]$$

Similar to the monoid operations, the monadic operations generalize the ordinary (indexed) writer monad operations by using the separation logic connectives.

As discussed previously in section 4.5, the above internal bind operation is hard to use in Agda. We again remedy this problem by exploiting the fact that the internal bind is equivalent in expressive power to the external bind (_=<<_) in combination with tensorial strength str over the separating conjunction $*$:

Figure 5.18: The external bind and strength for the writer monad.

▼

```
_=<<_ : ∀[ P ⇒ Writer W i₂ i₃ Q ] → ∀[ Writer W i₁ i₂ P ⇒ Writer W i₁ i₃ Q ]
str   : ∀[ Q ∗ Writer W i₁ i₂ P ⇒ Writer W i₁ i₂ (Q ∗ P) ]
```

Using the external bind we can again write our monadic computations using *do*-notation in Agda, which desugars in the usual way.

We can now instantiate Writer with Bytecode as the output type W, using the fact that Bytecode is an indexed monoid.[38] We will call this instantiation Compiler:

[38] The definition of Bytecode via IStar makes it trivially an indexed monoid.

```
Compiler : VarCtx → StackTy → StackTy → Pred Intf → Pred Intf
Compiler Γ ψ₁ ψ₂ = Writer (Bytecode Γ) ψ₁ ψ₂
```

The indices of the monad keep track of the typing of this bytecode output stream, so that we can ensure that we output type-correct instructions. The resource is instantiated to interfaces so that we can ensure that jumps are well-bound. By using the connectives from separation logic, their signatures express resource constraints that are upheld by the operations. This enables their use in computations that need to track resource use without requiring external proofs about their behavior.

To ease programming with Compiler, we define two derived monadic operations. Using tell, we define the operation code, which sends a single unlabeled instruction to the output. Using censor, we define the operation attachTo for labeling the first instruction of the generated output with the label ℓ passed as an argument:

Figure 5.19: Typed implementation of attachTo using the writer primitive censor.

▼

```
oneplus  : ∀[ P ⇒ Plus P ]
oneplus p = p •⟨ •-idʳ ⟩ nil

oneistar  : ∀[ R i₁ i₂ ⇒ IStar R i₁ i₂ ]
oneistar p = icons (p •⟨ •-idʳ ⟩ inil)

code   : ∀[ Import (Instr Γ ψ₁ ψ₂) ⇒ Compiler Γ ψ₁ ψ₂ Emp ]
code i = tell (oneistar (plain i))

attachTo : ∀[ Export (Own [ ψ₁ ]) ⇒ Compiler Γ ψ₁ ψ₂ P —∗ Compiler Γ ψ₁ ψ₂ P ]
attachTo ℓ = censor (mappend (oneistar (labeled ((oneplus ⟨$⟩ ℓ) •⟨ •-idʳ ⟩ (imports nop)))))
```

The first argument of censor of type:

$$\text{Bytecode}\ \Gamma\ \psi_1\ \psi_2 \multimap_* \text{Bytecode}\ \Gamma\ \psi_1\ \psi_2$$

is a function that transforms the output of the compiler computation and can own resources. We pass it the function mappend (..) that prepends a singleton bytecode sequence, consisting of a nop instruction labeled with the label passed to attachTo. To convert the label ℓ to a Labeling, we use the functorial map ⟨$⟩ on Export to lift the function oneplus that constructs a singleton labeling.

We also define the shorthand attach for outputting a label attached to a nop instruction:

```
attach : ∀[ Export (Own [ ψ1 ]) ⇒ Compiler Γ ψ1 ψ1 Emp ]
attach ℓ = attachTo ℓ ⟨ •-id′ ⟩ code (imports nop)
```

THE DEFINITION OF attachTo shows that label binders (and references for that matter) can be treated as first-class values using the linear typing discipline. It is useful to consider what happens under the surface here. Recall that binders are represented in a nameless fashion so that references to them are only given meaning by the relationship depicted by the interface composition witnesses. Hence, attaching the binder ℓ is only a meaningful operation if the interface compositions that occur in the output of attach, and between its output and the surrounding bytecode, are exactly the right proof terms. Yet, thanks to the abstraction of the logic, the programmer is not bothered with any of this. Instead, they are only concerned with satisfying the resource constraints that the type-checker enforces, which in this case means that they can only use ℓ once.

WE NOW PUT ALL INGREDIENTS we developed together to define an intrinsically-typed compiler from IMP to Bytecode. We focus on the compilation of expressions, which is done using:

```
compile_e : Exp Γ a → ϵ[ Compiler Γ ψ (a :: ψ) Emp ]
```

This signature expresses concisely the stack invariant of expressions—it says that the bytecode will operate in any stack ψ, and leave behind a single value whose type a matches the expression's type. The compiler is defined by pattern matching on the expression. As a warm-up exercise, let us consider the compilation of constants, which translate to a single push instruction:

```
compile_e (num x) = do                compile_e (var x) = do
  code (imports (push (num x)))         code (imports (load x))
```

The integration of the stack invariant in the type of the compiler, prevents us from writing bytecode that is not stack safe. Agda's type checker ensures that we meet the stack postcondition, and do not make any unwarranted assumptions about the input stack ψ. For example:

```
compile_e (num x) = do
  code (imports (push {! bool true !}))
  – error boolean != int
```

```
compile_e (bool b) = do
  code (imports {! swap !})
  – error _a_82 :: _b_83 :: _ψ_84 != ψ
```

The compilation of constructs that involve control flow is a little more involved as we have to generate labels. The compilation of conditionals if c then e_1 else e_2 is as follows:

```
compile_e (if c then e1 else e2) = do
  compile_e c
  let (lthen- •⟨ σ1 ⟩ lthen+) =  ∗-swap freshLabels
  lthen+                      ← ∗-id⁻ʳ ⟨$⟩ (lthen+ &⟨ •-comm σ1 ⟩ code (iftrue ⟨$⟩ lthen-))
  compile_e e2
  let (lend- •⟨ σ2 ⟩ labels)  =  ∗-rotate_l (∗-assoc_r (freshLabels •⟨ •-idˡ ⟩ lthen+))
  lthen+ •⟨ σ3 ⟩ lend+        ← ∗-id⁻ʳ ⟨$⟩ (labels &⟨ •-comm σ2 ⟩ code (goto ⟨$⟩ lend-))
  lend+                       ← ∗-id⁻ʳ ⟨$⟩ (lend+ &⟨ •-comm σ3 ⟩ attach lthen+)
  compile_e e1
  attach lend+
```

This code consists of the same eight operations as used in the untyped compiler in figure 5.1. The most significant difference in appearance arises due to the need to use tensorial strength $m \,\&\langle\, \sigma \,\rangle\, q$ to carry values across operations.[39] Here, σ witnesses separation between mp and q. Using strength, we build up a context of values that own resources, represented as a big separating conjunction. We use the laws of the separation logic (e.g., $*\text{-id}^{-l}$ and $*\text{-rotate}_l$) to simplify and reorder this context so that values appear in the right order. When using values, we have to provide the separation witness that relates the resource that they own to the resources owned by the context. This witness comes from the deconstruction of the monadic context on the left, or is trivial if the operation is resource neutral.

▲

Figure 5.20: Intrinsically typed compilation of conditional expressions.

[39] Recall that this notation stands for str $(m \bullet\langle\, \sigma \,\rangle\, q)$ and binds very loosely.

As an example, we consider the last three lines of the compilation. The first of these returns a context of two binding occurrences of labels *lthen+* and *lend+*, separated by (the interface composition) σ_3. We attach *lthen+*, and keep *lend+* in the context, to be attached to the end of the output. We do this with tensorial strength, using σ_3 as evidence that label to be attached is indeed separated from the context. In the last line we finally attach *lend+*, which leaves the context empty, and the resources balanced. If we omit the operation attach *lthen+*, and thus create a dangling reference, then Agda would reject the definition, because the resources that *lend+* owns were dropped.

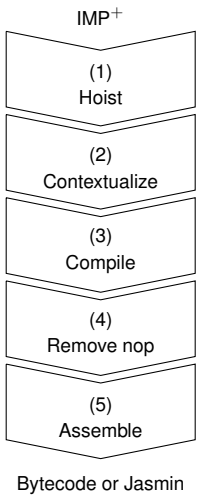
Compilation of the other expression constructs of IMP follows a similar pattern. The same is true for the compilation of statements, with the only significant difference being the type of the compiler, which obeys by a different stack invariant, leaving the stack exactly as it found it.

Because we want to use pattern matching and do-notation in Agda, we have to resort to the trick of programming with the external bind and tensorial strength, rather than entirely within the logic using the internal bind. It would be interesting to investigate whether it is possible to extend the syntax of the host language (i.e., Agda or Idris) to make the latter practical[40].

[40] Brady et al. 2012. "Resource-safe systems programming with embedded domain specific languages"

5.9 *An Intrinsically-Typed Compiler Backend in Agda*

Our intrinsically-typed compiler from IMP to bytecode (section 5.8) is part of a typed compiler backend pipeline that can be summarized using the figure in the margin. The compilation starts with typed IMP$^+$ where lexical variable binding is represented co-contextually using the co-de-Bruijn encoding[41]. The co-contextual typing is useful, because it allows hoisting (stage 1) declarations without weakening typed IMP$^+$ code to account for the additional binding. The hoisting itself is implemented in a monad that collects the binders. After hoisting, there are no more local variable declarations left, giving us co-contextually typed IMP. The *contextualize* transformation (stage 2) then eliminates the co-de-Bruijn encoding to contextually-typed IMP (whose syntax is in figure 5.16). A generic version of transformation 2 is given by McBride (2018).



[41] McBride 2018. "Everybody's got to be somewhere"

As shown in section 5.8, we then compile IMP to co-contextually typed bytecode (stage 3). The compiler inserts some unnecessary

nop instructions. We remove those unnecessary instructions using a simple intrinsically-typed bytecode optimization pass (stage 4). This transformation is pure, and implemented in 17 lines of Agda. Like compilation it requires no lemmas about the interface PRSA except the axioms of proof-relevant separation algebras. Bytecode optimizations are simplified greatly by representation that use labels, because unlike absolute or relative addresses, labels can be moved.

The final stage (stage 5) of our compiler backend is eliminating labels altogether, replacing them by absolute addresses of instructions. We implement this transformation in an intrinsically-typed style.[42] Alternatively, one can use our implementation of an unverified transformation to Jasmin code, which can be assembled to a Java class file and run by the JVM.

Our compiler pipeline lacks a frontend for parsing and typechecking IMP$^+$ programs. To test our compiler pipeline we have manually embedded a couple of IMP$^+$ programs in Agda, and used Agda's extraction to Haskell to obtain the generated bytecode.

Most of the code passes the Agda type checker under the `--safe` flag. Exceptions to this are the hoisting, contextualization, and assembly transformations. The former two do not pass the termination checker as implemented, and the latter uses Agda's rewriting mechanism[43] to automatically re-associate list concatenations when necessary. The compiler pipeline and the source, target, and intermediate languages are implemented in ~1700 lines of Agda.

[42] The key ideas of the transformation were mentioned in section 5.3.

[43] Cockx 2020. "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules"

5.10 Related Work

Co-contextual and principal typing. We faced the problem that the invariant of compilation to a low-level language with label binding was not preserved by the individual compiler operations. We addressed this problem by reformulating bytecode typing co-contextually. The co-contextual typings are *more principal*[44] than contextual typings, because the encapsulated knowledge about labels is restricted to be the minimal amount required to type the bytecode fragment. That is, they are a principal representative for many typings that can conceptually be obtained by weakening the context.[45]

From this perspective, we can think of our nameless and co-contextual encoding of label binding as a way to push the limits of what properties of our bytecode languages are principal. By going nameless, we were able to decide at the *outside* of a bytecode

[44] Jim 1996. "What are principal typings and what are they good for?"

[45] Our bytecode typing is not entirely principal. In particular, for every concrete bytecode fragment we assume that we know enough of the context to type the entire stack. This proved sufficient for compilation, where we always have enough knowledge about the stack. We also compile to JVM bytecode, which requires that the entire stack can be typed with a *monomorphic* type. This is different for some other typed low-level languages, such as typed assembly language (Morrisett et al. 1999a; Morrisett et al. 1999b), which support more principal typings for stacks via stack polymorphism.

fragment how labels defined *inside* of it relate to other labels. In other words, the nameless encoding is a *principal labeling*, where we assume less about labels that are used, and can use labels that are defined in more ways.

A key point of these principal typings is the *compositional* nature, which accommodates separate analysis, but can also help to define *total* functions on typed terms[46]. Examples of work that exploit this, knowingly or unknowingly, are plentiful. For example, there are lines of work on incremental type checking[47,48], and separate compilation[49,50]. Notably, the idea of treating exports and imports of names as opposites in order to give co-contextual accounts of global binding, is already somewhat present in the work on (not mechanically verified) co-contextual type checking by Kuci et al.[51], where references to globally defined names yield constraints that are removed at the top-level by corresponding declarations.

McBride[52] proposed the co-de-Bruijn encoding of lexical binding in a lambda calculus so that hereditary substitution on well-typed terms becomes a total function. In chapter 4, we gave a typing of linear references inspired by that encoding, incorporating the two roles of supply and demand in a state monad. The nameless model for labels builds on top of both these ideas. In retrospect, both are co-contextual reformulations of existing type-systems, resulting in more compositional typings.

The flip side of these more principal typings is that work is moved from the leaves of term typing to the nodes. This is visible in all the work cited above, and is also present in our work in the premises about interface composition. McBride[53] already introduces *relevant pairs* as the right notion of composition. In chapter 4, we identified this notion as the separating conjunction and extend it to a complete proof-relevant separation logic. In this chapter, we provided a new example of the applicability of all this logical structure, provide new data structures, and computational structures built on top of this logic, and also construct the modalities Export and Import that are specific to the instantiation of separation logic with interfaces.

Having identified this relation between co-contextual/principal typings of terms and proof-relevant separation logic, it would be interesting to apply this approach to other invariants that appear anti-compositional, and are thus hard to express intrinsically. Both to systematize the approach, and to design a better dependently-typed meta-language that can further simplify the definition of functions

[46] Wells 2002. "The essence of principal typings"

[47] Erdweg et al. 2015. "A co-contextual formulation of type rules and its application to incremental type checking"

[48] Kuci et al. 2017. "A co-contextual type checker for Featherweight Java"

[49] Jim 1996. "What are principal typings and what are they good for?"

[50] Ancona et al. 2004. "Even more principal typings for Java-like languages"

[51] Kuci et al. 2017. "A co-contextual type checker for Featherweight Java"

[52] McBride 2018. "Everybody's got to be somewhere"

[53] McBride 2018. "Everybody's got to be somewhere"

that utilize the framework, like our compiler.

Intrinsic verification of compilers. The idea of developing type-safe compilers in a dependently typed language is first described by McKinna et al.[54]. They work in Epigram[55] and show that it indeed feasible to define an intrinsically typed compiler for a simple expression language to typed instructions. A key idea is the indexing of bytecode by two stack types, which we successfully applied again in our JVM bytecode language.

Their low-level code is not linear. That is, control-flow is represented using an instruction IF c_1 c_2, where c_1 and c_2 are again blocks of instructions. Hence, their compiler does not need labels. We focused on linear bytecode because compilers must eventually output linear code for the machine. They also prove that their intrinsically typed compiler is functionally correct with respect to an interpreter of the expression language. Defining the dynamic semantics and stating and proving functional correctness will be more difficult for our source and target languages, because neither is normalizing.

[54] McKinna et al. 2006. "A type-correct, stack-safe, provably correct, expression compiler"

[55] McBride 2004. "Epigram: Practical Programming with Dependent Types"

Abel[56] presented a well-typed compiler that targets a typed representation of control-flow graphs. In his intermediate language, control-flow is reduced to two primitive constructs: *join* points and *looping* points. Unlike bytecode this is not a flat language, which allows labels to be treated like lexical variables in e.g., the well-typed syntax of STLC. A second compiler pass (linearization) must then translate control flow graphs to a flat bytecode language, but their implementation of this pass has not been published. The author does note that the full verification requires reasoning in the sublist category. We have shown that we can avoid such reasoning by working with nameless binding in an embedded linear language.

[56] Abel 2020. "Type-preserving compilation via dependently typed syntax in Agda"

McBride[57] showed how to intrinsically verify *functional* correctness of a compiler for a small arithmetic expression language by indexing syntax by their semantics. This is extended by Pardo et al.[58] with top-level variables, and sequenced and looping assignments to those variables. They also compile via an intermediate language with sequencing and loops, and output low-level bytecode with relative jumps. Although the intermediate language is indexed by a semantics, the low-level target language is not. Instead they give this semantics extrinsically, because it is unclear whether it is possible to

[57] McBride 2012. "Agda-curious?"

[58] Pardo et al. 2018. "An internalist approach to correct-by-construction compilers"

give a "syntax-directed" semantics for the low-level language with jumps.

Extrinsic compiler verification. There is a lot of prior work on verifying correctness results of compilers and compiler optimizations in proof assistants. Compilers often compile via various intermediate languages to simplify transformations and their verification[59,60]. *Control-flow graphs* and *continuation-passing style* are commonly used intermediate representations in compilers (see for example Appel[61], Morrisett et al.[62], Chlipala[63], Guillemette et al.[64], and Bélanger et al.[65]). Such intermediate languages are a more suitable level of abstraction for expressing high-level compiler optimizations. After applying the optimizing transformations, these higher-level representations of control-flow are eliminated in favor of jumps and labels in a low-level machine language. In a well-typed version of such a multi-stage compiler, our compilation monad would only be used for this pass, which is usually called *linearization*. The problem that verifying label binding requires additional reasoning in the compiler definition, is not an issue in these works, because all the verification is done extrinsically. Good extrinsic proof automation can be an alternative approach to avoiding manual proof overhead.

In extrinsic verification, two other methods have been used to denote jump targets: instruction addresses (e.g., in the machine language of CakeML[66]) and instruction offsets (e.g., in the Jinja compiler[67]). Although both are candidates for intrinsically-typed bytecode representations, they cannot be used in the compilations in this chapter, which output forward jumps before recursively compiling the instructions that are targeted. Both encodings require computing the output up-to the jump target first, so as to know the address of the target. In addition, it is difficult to write bytecode transformations using absolute or relative addresses. Even our nop removal transformation would require shifting jumps throughout the program. In label-based representations this is much simpler, because labels can be moved.

The design of more expressive type systems for assembly languages (e.g., Morrisett et al. (1999b) and Crary[68]) is an important, but largely orthogonal research direction. The same is true for research into semantic type systems for machine languages, and verification techniques that reduce the trusted computing base[69].

[59] Leroy 2009. "Formal verification of a realistic compiler"

[60] Kumar et al. 2014. "CakeML: A verified implementation of ML"

[61] Appel 2006. "Compiling with continuations"

[62] Morrisett et al. 1999b. "From system F to typed assembly language"

[63] Chlipala 2007. "A certified type-preserving compiler from lambda calculus to assembly language"

[64] Guillemette et al. 2008. "A type-preserving compiler in Haskell"

[65] Bélanger et al. 2015. "Programming type-safe transformations using higher-order abstract syntax"

[66] Kumar et al. 2014. "CakeML: A verified implementation of ML"

[67] Klein et al. 2006. "A machine-checked model for a Java-like language, virtual machine, and compiler"

[68] Crary 2003. "Toward a foundational typed assembly language"

[69] Appel 2001. "Foundational proof-carrying code"

Linear meta-languages. We have constructed a shallowly *embedded* language that tracks resource usage. Another approach is to use a meta language with *built-in* support for resources, like linear Haskell[70], Idris[71] (where Idris 2 implements quantitative type theory (QTT)[72]), or Granule[73]. To be able to use such a language instead of our separation logic, the language needs to support not only user-defined resources, but also *proof-relevant* resources. Our logic not only hides the accounting at the time of type-checking the compiler, but also the construction of a proof term that exists at runtime. We implement the primitives (e.g., freshLabels) by explicitly making use of the fact that the logic is a shallow embedding. When we eliminate labels in the assembler, we again poke through the abstractions of the logic, and compute directly on the proof terms. To the best of our knowledge there exist no languages with built-in support for resources such as our interfaces. Granule appears to come the nearest to this, as its theory permits user-defined resources. The current implementation, however, does not. A resource in Granule and QTT must be a semiring with functional operations[74], and thus does not support our PRSA resources. Despite the fact that an expressive enough language does not yet exist, these lines of work are a promising path towards a more suitable dependently typed meta language.

[70] Bernardy et al. 2018. "Linear Haskell: Practical linearity in a higher-order polymorphic language"

[71] Brady 2013. "Idris, a general-purpose dependently typed programming language: Design and implementation"

[72] Atkey 2018. "Syntax and semantics of quantitative type theory"

[73] Orchard et al. 2019. "Quantitative program reasoning with graded modal types"

[74] Orchard et al. 2019. "Quantitative program reasoning with graded modal types"

5.11 Conclusion

We presented the intrinsically-typed compilation of a small language with structured control flow to typed bytecode, giving rise to a compiler that is type correct by construction. The key problem we faced was the fact that label well-boundness appeared to be an anti-compositional, whole-program property. Our first key idea to solve this problem is to reformulate label well-boundness using a nameless and co-contextual representation, which turns it into a compositional, local property. Our second key idea is to abstract over co-contexts and their compositions using a proof-relevant separation logic. As a result, our intrinsically-typed compiler, as well as the operations that support it, contain little manual proof work and mirror their untyped counterparts.

PART II
DECLARATIVE SPECIFICATION OF STATIC SEMANTICS

6 Sound Type Checkers from Typing Relations

"Quis custodiet ipsos custodes?"
— Decimus Junius Juvenalis, from *Satires*

In chapter 1, we asked the question how a language developer should ensure that their language's type checker is sound with respect to the specification of the static semantics. In an ideal world, programming language designers should not have to deal with accidental complexity when defining and implementing languages. Some aspects of language design are already close to realizing this ideal. For example, parser generators make it possible to obtain parsers from declarative grammar specifications, thus abstracting over the accidental complexity of implementing parsing. There should be similar support for generating implementations of type checkers from their declarative specifications.

The variety of language features found in real-world languages presents many challenges in the way of this ideal. This chapter focuses on the challenges presented by name resolution, an aspect common to all programming languages. Many language features found in actual languages interact with name resolution. Modules, imports, classes, interfaces, inheritance, overloading, and type-dependent member access to objects and records are a few examples that are commonplace. Implementing type checkers for languages with such features is complicated because the use of names in programs causes dependencies between type-checking tasks, and requires that the *construction* of symbol tables and type environments is interleaved with *querying* those data structures. Evaluating a query too early may result in an unstable answer—i.e., an answer that is invalidated by subsequent additions to the environment or symbol table. A wrong

The technical material of this chapter was published in A. Rouvoet, H. Van Antwerpen, C. B. Poulsen, R. Krebbers, and E. Visser (2020a). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications". In: *Proceedings of the ACM on Programming Languages* 4.Object-Oriented Programming Systems, Languages & Applications (OOPSLA), 180:1–180:28. DOI: 10.1145/3428248.

answer can have far reaching consequences, either compromising the soundness of the type checker, or later requiring backtracking on an arbitrary amount of work that depends on the wrong answer.

Consider, for example, the following Scala program:

```
object M {                                  1
  object B { ... }                          2
}                                           3
import M.B;                                 4
object A {                                  5
  import B._;                               6
  ...                                       7
}                                           8
object B { ... }                            9
```

◀

Figure 6.1: Forward reference to a definition that shadows a previous definition.

A type checker working its way forward through the program would initially resolve `import B._` to the imported object `M.B`, and type check the remainder of the body of `A` under the resulting environment. If only then it encounters the local declaration of `B` on line 7, it needs to redo the type checking of the body of `A` because the local definition shadows the earlier imported declaration.

To avoid this, the interleaving chosen by the type checker must ensure that *query resolution is stable*—i.e., that answers to queries that consult the symbol table are not invalidated by subsequent additions to the environment or symbol table. This can be a non-trivial scheduling problem because environment and symbol table construction can also *depend* on answering queries.

Languages often have many features that interact with name binding and disambiguation, and as a consequence it can be difficult to construct schedules that guarantee query stability. The following Scala program shows for example how classes and inheritance interact with name resolution:

```
class A extends B.D {                       1
  def g:Int = f                             2
}                                           3
object B extends C {}                       4
class C {                                   5
  class D {                                 6
    def f:Int = 1                           7
}}                                          8
```

◀

Figure 6.2: Inheritance in Scala, interacting with nested class declarations.

In this program, the `f` on line 2 resolves to the `def f` on line 6; but for this resolution to succeed, the qualified reference `B.D` on line 1 must first have been resolved to the `D` on line 4, to make the bindings in class `class D` reachable from the body of `class A`. Resolving `B.D` in turn depends on: (1) resolving the `B` in `B.D` to `object B` on line 4; and (2) resolving the `C` in the extends clause for `object B` on line 4 to the `C` declaration on line 5.

Determining these dependencies requires a good understanding of the binding and disambiguation rules of a language. The type checking algorithm must take all these dependencies into account, so that names are only resolved once all information that is relevant to their resolution is collected. If this is the case, then the result of name resolution is stable. Type checker implementations use various strategies for stratifying or scheduling the collection and querying of name binding information. Every type checker must, implicitly or explicitly, solve this scheduling problem. For example, Haskell's binding restrictions ensure that binding collection and resolution can be separated into static passes over the program, whereas Scala and Rust require type-dependent name resolution, which requires interleaving type checking and name resolution. A key property of sound strategies is that names are only resolved after all the relevant information has been collected.

The concrete strategies are irrelevant for understanding and reasoning about the underlying type system, but crucial to a correct implementation of the type checker. This tension between implementation and specification is felt by language designers. For example, the Rust language developers write the following about specifying name binding in the language:[1]

> Whilst name resolution is sometimes considered a simple part of the compiler, there are some details in Rust which make it tricky to properly specify and implement.

[1] Rust developers 2020b. "About Rust's name resolution"

And in reply to changes to the design and implementation of name binding, a contributor states:[2]

> I'm finding it hard to reason about the precise model proposed here, I admit. I wonder if there is a way to make the write up a bit more declarative.

[2] Rust developers 2020a. "About declarative specifications of Rust's name resolution"

A more declarative specification should allow reasoning about name binding without having to rely on an understanding of the opera-

tional details such as the scheduling of name and type queries. But if we want to obtain type checkers from these declarative specifications, we need to be able to automatically construct sound schedules. In this chapter we give a language independent explanation of necessary and sufficient conditions to guarantee stability of name and type queries during type checking. We use this to make declarative type system specifications executable as type checkers for the specified language. Using this approach, we can guarantee that the resulting type checkers are *sound* with respect to the formal declarative semantics of the specifications, as well as *confluent*. These important properties of type checkers are proven once-and-for-all for languages specified using our formalism, rather than on a language-by-language basis.

Problem. We start from a specification of an object language's static semantics in the meta-language Statix[3]. Language specifications in Statix are given by typing rules, written as predicates on terms, types, and a *scope graph*[4]. Scope graphs generalize language specific notions of type environments and symbol tables. A distinguishing feature of Statix are its *scope graph assertions and queries*, which can be used to give high-level specifications of name resolution. These assertions can express fine-grained name resolution rules, which enable high-level specification of, for example, shadowing rules of Java and Scala.

[3] Van Antwerpen et al. 2018. "Scopes as types"

[4] Neron et al. 2015. "A Theory of Name Resolution"

The problem we face is to derive a type-checker from a Statix specification. Statix's scope graph assertions and queries make it possible to give high-level specifications of name binding, but, at the same time, make the problem of deriving these type checkers more difficult. In particular, we have to solve a generalized version of the scheduling problem described above. That is, we need a *general* characterization of the conditions under which it is sound to query symbol tables and type environments during type checking. We then need to derive a type checker from a Statix specification in such a way that these conditions are always satisfied.

The general approach to deriving type checkers from Statix specifications is already sketched by Van Antwerpen et al.[5], who provide a Java implementation. They explain the problem with unsound name resolution when queries answers are unstable, and they claim that their implementation implements a sound strategy. This strategy, however, is only informally described, and lacks evidence of its

[5] Van Antwerpen et al. 2018. "Scopes as types"

soundness.

This chapter addresses both those deficiencies by formalizing the derivation of type checkers from Statix specifications, and proving soundness. Our formalization of the operational aspects of Statix revealed that the Java implementation is, in fact, not confluent[6], which we address in this chapter by refining the scope graph primitives. Confluence is an important property of Statix because its implementation is non-deterministic. It ensures that the solver does not have to backtrack on evaluation order. In order to formalize the soundness and confluence results, we develop a theory around the novel concept of *critical edges in scope graphs*. We believe that this concept is a useful device in both the design of languages, and the implementation of their type checkers. We also hope that the formalization of the operational semantics of Statix makes it feasible to port the novel ideas of Statix about the high-level specification of name binding to other formalisms and type checker implementations.

[6] The Java implementation answers name resolution queries with lists of results. This is convenient, because it allows results to be treated as terms. The formalization revealed, however, that query answers are really sets and cannot be sorted in a stable manner. This motivates why our proposed core language contains primitives for expressing constraints quantifying over elements of a set (see figure 6.9). These primitives are picked carefully to ensure that a non-deterministic solver remains confluent.

Approach. To enable this formalization, we first introduce Statix-core. This core language refines and simplifies the previous formulation of the Statix meta-language. The declarative semantics of Statix-core is similar to the declarative semantics of Statix, and explains what are valid type derivations of a specified language. In other words, it explains when a *given* object-language program, together with a type assignment and a scope graph model of its binding, *satisfies* the specified static semantics of an object language.

We equip this refined core of Statix with a novel small-step operational semantics. This operational semantics takes a specification and an object-language program, and then computes a type assignment and a scope graph, thus fulfilling the task of a type checker for the object language. The key question of this chapter arises when we try to define how queries in Statix compute. What are the conditions that ensure that the answer to a scope graph query is stable under future additions to the scope graph model of binding in the program?

To make the condition for query answering precise, we introduce the new idea of *critical edges* for a query in a scope graph extension, precisely characterizing missing dependencies of the query. Conceptually, query answers that are computed in a partial scope graph are stable if recomputing the answer in a complete model of the program yields the same result. We will show that it is safe

to answer a query in a partial graph $\mathcal{G}$ when the complete model contains no critical edges for the query with respect to $\mathcal{G}$.

The absence of critical edges in the complete model can in practice not be checked by a type checker because it requires knowing the complete model of binding upfront. We solve this by weakening the condition to a *sufficient* condition that can be checked. We then impose a well-formedness judgment on Statix-core specifications to also make this tractable in practice. Specifically, typing rules must have *permission to extend* a scope in the scope graph to be able to make assertions on the scope graph. In practice this means that although scopes can be queried from anywhere, they can only be extended locally with new binding information.

We prove that the operational semantics of Statix-core using the weaker sufficient condition is *sound* for well-formed specifications—i.e., it computes a type assignment and scope graph model that satisfy the specification. Importantly, and in contrast to the original implementation of Statix[7], the non-deterministic operational semantics can also be proven *confluent* for the refined Statix-core language. The confluence argument again uses critical edges to reason about stability of query answers.

[7] Van Antwerpen et al. 2018. "Scopes as types"

We implement, in Haskell, the operational semantics and the static analysis that checks if all rules have sufficient permissions to extend scopes. We give specifications of subsets of Java and Scala in Statix-core (extended with recursive predicates). Using these specifications we also *test* soundness of the reference implementation against the Java and Scala type checker. These case studies provide evidence of the expressiveness of Statix as a formalism, and show that the well-formedness restriction does not prohibit specifications of complex, real-world binding patterns.

In summary, the contributions of this chapter are:

- Statix-core (section 6.2), a constraint language with built-in support for scope graphs, which distills and refines the core aspects of the Statix language and its declarative semantics.
- A semantic characterization of name resolution query answer stability in terms of *critical edges* in an incomplete scope graph (section 6.4).
- An operational semantics for Statix-core (section 6.3 and section 6.4) that schedules name resolution queries such that query

answer stability is guaranteed, thereby allowing language designers to abstract from the accidental complexity of implementing name resolution.

- A proof that the operational semantics of Statix-core is sound with respect to the declarative semantics of Statix-core (theorem 1 and theorem 11). The key that enables this proof is a type system for Statix-core (based on *permission to extend* a scope) and the scheduling criterion that is built into the operational semantics of Statix-core (based on an over-approximation of critical edges).
- MiniStatix, a Haskell implementation of Statix-core extended with (recursive) predicates. The implementation infers whether specifications have sufficient permissions to extend scopes, and can type check programs against their declarative language specification.
- Three case studies (section 6.5) of languages specified in MiniStatix: (1) a subset of Java that includes packages, inner classes, type-dependent name resolution of fields and methods; (2) a subset of Scala with imports and objects; and (3) an implementation of the LMR[8] module system that is similar to the one in Rust. The case studies demonstrate the expressive power and declarative nature of Statix-core, and test the approach against the reference type-checkers of Java and Scala.

[8] Van Antwerpen et al. 2016. "A constraint language for static semantic analysis based on scope graphs"

6.1 *Specifying & Scheduling Name Resolution*

Programming languages with modules or objects (e.g., ML, Java, C♯, Scala, or Rust) use very different name resolution rules than languages with only lexical static binding. For example, the static semantics of non-lexical static binding, such as accessing a member of an object `o.m`, is to resolve the name `m` not in the *local* (lexical) scope, but in a *remote* scope (in this case the inner scope of the class declaration that corresponds to the type of the reference `o`). Similarly, a name in Scala or Rust is not always resolved in the lexical scope, but sometimes in an explicitly imported module or object scope, whose definitions may be declared in a very different part of the program.

These richer scoping constructs lead to more subtle resolution and disambiguation rules. Scala, for example, applies different scoping rules for names defined in the lexical scope (which can be

forward referenced) compared to names that are imported (which cannot). Scala also applies different precedence rules depending on whether an imported name is explicitly listed, or caught by a wildcard. Precedence rules are often incomplete, in the sense that overlapping names sometimes lead to ambiguous uses. This requires more information to be available in environments.

These aspects make it more difficult to both *specify*, and *implement* static semantics. In this section we discuss both specification and implementation. We first discuss the role of name binding in the specification of static semantics, and how Statix as a formalism enables the high-level specification of the above mentioned features. We then discuss how name binding features contribute to a scheduling problem for type checkers. Finally, we show how the innovative features of Statix impact this scheduling problem. We will argue that there are two sides to this. On the one hand, these features make the scheduling problem more difficult because value dependencies are less explicit. On the other hand, the high level specification of binding in Statix provides a semantic tool to think about the scheduling problem and recover a provably sound schedule: critical edges. We end this section with an overview of how we use critical edges to address the scheduling problem for Statix.

NON-LEXICAL STATIC BINDING can easily complicate a specification of static semantics, harming conciseness, understanding, and maintenance of the static semantics rules. Conventional typing rules use *type environments* (or, typing contexts) to propagate binding information through a program. Type environments are appropriate and easy to use in the specification of static semantics for languages with only lexical binding because lexical binding follows the nesting structure of the AST. This is not the case for languages with non-lexical static scoping, where binding information may flow through references (e.g., module imports), or against the nesting structure of the AST (forward references).[9]

[9] Hedin 2000. "Reference Attributed Grammars"

To demonstrate the issues that arise in language specification, we consider a simple Scala program. The program in figure 6.3 is a well-typed Scala program with two methods in an object `o` that mutually refer to one another.

```
object o {
  def f:Int = g;
  def g:Int = f
}
```

◀

Figure 6.3: Mutual binding in Scala.

To specify the static semantics of such a list of mutually recursive definitions, we can follow the style of the ML specification[10], which uses rules of the form $C \vdash e \Rightarrow E$, with C the type environment of the phrase e, and E the context generated by the phrase e. The context C is downward propagating, whereas E is upward propagating. We obtain the following rules for block definitions:

[10] Milner et al. 1997. "The Definition of Standard ML, Revised"

$$\textsc{T-Body}\ \frac{E + E' \vdash bs \Rightarrow E'}{E \vdash \{\ bs\ \} \Rightarrow E'} \qquad \textsc{T-Seq}\ \frac{E \vdash b \Rightarrow E' \qquad E \vdash bs \Rightarrow E''}{E \vdash b;bs \Rightarrow (E' \sqcup E'')} \qquad \textsc{T-Def}\ \frac{E \vdash e : T}{E \vdash (\texttt{def}\ f\ :\ T\ \texttt{=}\ e) \Rightarrow \{f : T\}}$$

▲

Figure 6.4: Typing of mutual binding using environments.

Name resolution behavior is the result of the way environments are combined in the different rules. The mutually-recursive behavior of the block is visible in rule T-Body, which *updates* the type environment with the aggregated binding that has propagated upwards from the block. The combination operator $+$ in the premise of T-Body updates the environment such that it shadows bindings in E that are also in E'. The disjoint union $\sqcup$ in the conclusion of T-Seq merges the environments produced by the definitions in the sequence, and enforces that the names do not overlap. We can see in this example that environments play two roles in these rules: to *aggregate* binding information from the program, and to *distribute* it throughout the program. Aggregation ties back into distribution at the scope boundary.

The update and disjoint union of environments are examples of bookkeeping operations that encode high-level binding concepts: shadowing and disallowing duplicate definitions respectively. Similarly, the 'cycle' in environment aggregation and distribution *encodes* mutual recursion. Encoding this using environments is a relatively small matter here, due to the limited number of rules and binding features to take into account. This becomes increasingly more difficult when we add language features that interact with binding and that require more sophisticated disambiguation.

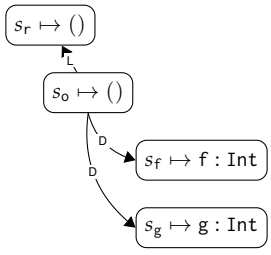
In particular, non-lexical static binding complicates matters significantly: the definitions in figure 6.3 are not just locally in scope, but can be accessed from remote use sites, either qualified with the object o, or unqualified after importing object o. The potential for remote use significantly increases the required effort for aggregating and distributing binding facts. To look up the structure of modules and classes, we may want to refer to a symbol table. Thus we have to explain through our typing rules how declarations generate unique entries in this symbol table. This requires aggregating all the entries to the root of the program. For the purposes of disambiguation, we may also need more structure in the environment. In Scala for example, we need to look beyond the closest matching binding because additional binders in outer scopes may make a reference ambiguous.

We argue that bookkeeping of environments is not a high-level means for expressing name resolution concepts of languages like Scala. Consequently, it is both unnecessarily hard to define rules that express the right semantics, and unnecessarily difficult to understand the high-level concepts from the written rules. Previous work proposes Statix[11] to address this problem. In section 6.2, we discuss the concepts of Statix. We will show how Scala's name resolution rules can be understood using scope graphs, and made precise using Statix rules.

[11] Van Antwerpen et al. 2018. "Scopes as types"

THE PROBLEM OF AGGREGATING AND DISTRIBUTING binding information is addressed by Statix using two ideas: (1) item scopes have independent existence and can be passed around, which allows extending scopes without the need for explicit aggregation, and allows remote access without explicit distribution; and (2) shadowing behavior is specified at the use site, allowing definitions to simply assert the scoping structure without having to anticipate all possible uses. To achieve this, Statix typing rules are predicates on terms and an *ambient scope graph*. Nodes in the graph represent scopes and binders, whereas (labeled) edges are used to represent (conditional) scope inclusion. Nodes contain a data term that can carry the information of a binder.

The binding of the program in figure 6.3 can be summarized as the scope graph in figure 6.5. We write $s \mapsto t$ for a node with identity s and data term t. The nodes s_R and s_o represent the root scope and the object scope respectively. The latter is a lexical child



▲

Figure 6.5: Scope graph for figure 6.3.

of the former, indicated by the L-edge. The object scope contains two declarations, indicated by the two D-edges to declaration nodes, whose data terms `f : Int` and `g : Int` contain the usual information about the binders.

Previous work has shown how scope graphs can be used to model many binding structures[12,13,14]. The fact that this particular scope graph models the binding of the given program, is made formal through a number of Statix rules, together with the declarative semantics of Statix. We give the required rules here using the Statix-core syntax, so that we can informally discuss how Statix constraints address the problems with declarative specification of binding using environments explained above. We will explain the formal syntax and declarative semantics of Statix-core in section 6.2.

[12] Neron et al. 2015. "A Theory of Name Resolution"

[13] Van Antwerpen et al. 2016. "A constraint language for static semantic analysis based on scope graphs"

[14] Van Antwerpen et al. 2018. "Scopes as types"

The Statix-core counterparts to the ML-specification style rules for the mutual binding in figure 6.3 are as follows:

$$\frac{(\nabla s' \mapsto ()) * (s' \xrightarrow{\llcorner} s) * (s' \vdash bs)}{s \vdash \{\ bs\ \}}\ \text{T-Body} \qquad \frac{(s \vdash b) * (s \vdash bs)}{s \vdash b;bs}\ \text{T-Seq}$$

$$\frac{(s \vdash e : T) * (\nabla s' \mapsto (f : T)) * (s \xrightarrow{D} s') * \mathsf{noDups}(s, f, s')}{s \vdash (\texttt{def f : T = e})}\ \text{T-Def}$$

Figure 6.6: Typing rules using Statix-core constraints.

The Statix specification consists of constraint rules, which define that the typing judgment in the conclusion holds if the constraints in the premises hold. The phrases are typed in a lexical scope s, written suggestively as $s \vdash t$.[15] Premises are separated using conjunction ($*$). The fact that a block introduces new scope is expressed in the rule T-Body by asserting a scope s' in the scope graph (using $\nabla s' \mapsto ...$), connected to the lexical parent by an L-edge (using $s' \xrightarrow{\llcorner} s$). The declarations are asserted similarly in the rule T-Def using a D-edge.

The first notable difference with the ML-style rules is that the Statix rules have no upward propagating context for aggregating binding. This is unnecessary because of the reference semantics of scopes in Statix rules. The rule T-Def can directly assert the structure that a definition induces in the ambient scope graph. Because the scope graph is a global model of binding, this structure does not need to be explicitly aggregated or distributed.

[15] The form of this typing judgment is not enforced in Statix rules—i.e., Statix predicates do not have to be defined exclusively over *AST* terms and can have multiple scope arguments.

The second difference is in the way that lexical shadowing is specified. Rather than encoding this disambiguation rule using environment update in T-Body, the Statix-core rule only witnesses the structure of the scope graph model. Disambiguation is expressed directly in the rule for typing variables. We postpone the discussion of scope graph *queries* that fulfill this purpose until section 6.2. For now it suffices to know that variable lookup works by finding minimal paths in the scope graph. Shadowing can be expressed by using a lexicographical path order where $\mathsf{D} < \mathsf{L}$.

The third difference is that the rule T-Seq is a completely binding-neutral rule. The fact that definitions should be unique in their scope, is expressed directly as a premise $\mathsf{noDups}(...)$ on the rule T-Def, rather than being encoded in the way that sequencing aggregates binders. We leave the predicate abstract, but it is specified using a graph query in the declaration scope.

Specification of languages with rich, non-lexical name binding features is complicated when using environment-based typing rules. Statix provides a general formalism that allows concise specification of these languages, by removing the concerns of aggregating and distributing binding information from the typing rules.

We now turn to the problem of writing a type checker based on a specification of static semantics, focusing on the difficulties surrounding name binding features. We will argue that type checkers face a *scheduling* problem in constructing the relevant environment and symbol table (or scope graph) to be able to type the names used in a program. Consider again the typing rules in figure 6.4. A type checker arriving at the block faces the problem that the downward propagating input environment is constructed from the upward propagating output environment. For this reason, the type checker needs to be *staged*: it first needs to aggregate the binding from the block, before it can type check the expressions in the right environment. This simple example demonstrates how name binding induces dependencies between tasks in a type checker. Name resolution (and thus type checking) is only sound with respect to the typing rules if queries are only executed after all relevant information has been aggregated.

The binding features of a language determine how difficult it is to find a sound schedule. A language with forward references requires a schedule in which binding aggregation happens before

querying. In our example, the schedule can be entirely static: one can always collect all definitions before typing their bodies. First class modules and type-dependent name resolution require more dynamic scheduling. For example, the resolution of a member name `m` in a Java or Scala expression `e.m(...)` requires the type of `e`. Typing `e` can in turn depend on all kinds of name resolution and type-checking, so that name resolution cannot be statically stratified.

When language developers implement a type checker for a given language, they either implement such a statically stratified schedule as a number of fixed type-checking passes, or implement a method that schedules type-checking tasks dynamically (even if the scheduling is simply 'on demand'). Soundness of the implemented approach is judged by the language developers. Our goal is to *automatically* obtain sound type checkers from typing rules, and therefore we need a *systematic* approach to solving the scheduling problem.

In section 6.1 we arrived at a sound schedule for the typing of mutually recursive binding simply by lazily following the *demand* for dependencies. These dependencies are explicit in the environment-based rules of figure 6.4. In languages with more complex scope and disambiguation rules, the dependencies of name resolution are not as easy to determine. We have argued that environment-based rules are difficult to specify for such languages. Ensuring that those rules can be evaluated on demand puts additional requirements on the rules, making it even more difficult to write the specification.[16] (This is a known problem with canonical attribute grammars. We compare in depth to attribute grammars in section 6.6.) By decoupling scope from binding and name resolution rules in those scopes, Statix rules can specify complicated languages without regard for dependencies. As a result, more work is required to reconstruct the dependencies and a sound schedule from the rules.

We illustrate this with the following Scala program, combining mutually recursive definitions and imports.

[16] Boyland 2005. "Remote attribute grammars"

```
object o {
  def f:Int = g;
  import n._;
  def g:Int = h
}
object n {
  def h:Int = 42;
}
```

◀

Figure 6.7: Scala example with mutual binding and imports. Note that the imported names from `n._` are only in scope in the part of the block *after* the import statement.

The semantics of Scala are such that the definitions in an object are mutually recursive, allowing the forward reference g, while imports are sequential, only allowing references to the imported name h after the import statement. Local definitions have precedence over names imported in the same block, regardless of the order in which the definitions and imports appear in the program.

The scoping structure of our example is modeled with the scope graph shown in figure 6.8.

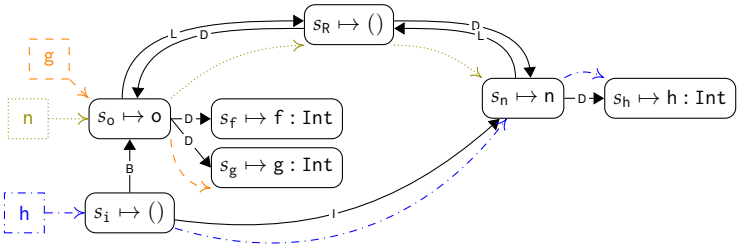


Figure 6.8: Scope graph corresponding to the program in figure 6.7. Colored nodes and paths only illustrate references and their resolution and are not part of the model.

The colored dashed/dotted boxes indicate references of a name (g, n, and h) appearing in a certain scope, with colored paths indicating how these references are resolved. The elements in color are *not* part of the scope graph, but only drawn for illustrative purposes. The definitions f and g are declared in the object scope s_o. The set of outgoing edges from a scope is *not ordered*. An imposed order between outgoing edges is instead modeled by chaining scopes. For example, because imports are treated sequentially in Scala, the import statements induces a scope s_i, connected to the previous import or object scope using a B-edge. The import makes more declarations accessible, which is modeled in the scope graph by an I-edge to the scope s_n of object n. The forward reference g resolves to the definition in the same scope. The reference to h reaches the imported name via the B-edge and the outgoing I-edge.

Name resolution is specified in terms of queries on the scope graph, which specify *reachability* and *visibility* of declarations in terms of a regular expression and an order on paths, respectively (section 6.2). In this Scala subset, a declaration is reachable if it can be found in the scope graph via a path that matches the regular expression $\mathsf{B}^*(\mathsf{LB}^*)^*\mathsf{I}^?\mathsf{D}$. One can check that all the colored paths indeed match the regular expression.

During type checking, the scope graph is constructed from an initial empty graph, by adding more and more scopes and edges, until the graph is a complete model of the binding and scoping structure in the program. Name resolution is finding least reaching paths in the scope graph. Although conceptually simple, difficulty arises because scope graph construction can depend on resolving queries as well as the other way around. This is the case for imports, where the I-edge depends on resolution of the named import. In general, even determining whether there is an edge *at all* can depend on name resolution. This means that scope graph construction *must* be interleaved with query evaluation.

This raises the following concrete scheduling problem: given a scope graph query, a partial scope graph, and a partially satisfied type specification, is it sound to evaluate the query now or should it be delayed? Conceptually, the answer is 'yes, it is sound' if the answer to the query in the current partial model is the same as the answer in a complete model. The answer is 'no, delay' if the complete model contains additional binding information that is relevant to the query at hand.

To specify what information is relevant, we introduce the notion of *critical edges* for a query in a model with respect to a partial scope graph. An unstable resolution answer means that a resolution path that is valid in the model graph is not yet a valid path in the partial graph because some part of the final graph is missing. A *critical edge* of a query is an edge along a resolution path in the model that is not present yet in the partial graph, but whose source node is present. We can think of critical edges as the root cause of instability, as they are the *first* missing step in a resolution path in the model. Whether an edge is critical is determined based on the regular expression that expresses reachability, which exactly demarcates the part of the scope graph that will be searched.

Because the complete model is yet unknown, we cannot directly identify missing critical edges. Instead, we look ahead at the remaining type checking problem to determine whether any critical edges are still missing. In general, precise determination may require arbitrary type checking, which would lead to a backtracking implementation. Instead, we approximate critical edges as *weakly* critical edges, whose absence can be determined without backtracking. We show that our approximation is sound for a subset of Statix

specifications. Importantly, we can statically determine if a specification is in this subset using a type analysis that we formalize as *permission-to-extend*.

6.2 *Statix-core: A Constraint Language*

In this section we introduce Statix-core, modeling the essential ingredients of Statix[17]: a framework for the declarative specification of type systems. Statix specifications have a precise declarative semantics that specifies which scope graphs are models of the specification. They do not have a formal operational semantics that can be used to find a model for a given program if it exists. Such an operational semantics requires a sound scheduling strategy for name and type resolution.

[17] Van Antwerpen et al. 2018. "Scopes as types"

First, we formally introduce scope graphs together with a concise presentation of the resolution calculus[18,19]. We then present the syntax and declarative semantics of Statix-core. Subsequently, in section 6.3 and section 6.4, we present the sound operational semantics using a general delay mechanism for queries based on critical edges.

[18] Neron et al. 2015. "A Theory of Name Resolution"

[19] Van Antwerpen et al. 2016. "A constraint language for static semantic analysis based on scope graphs"

Statix-core is a constraint language extended with primitives for *scope graph assertions and queries*. The assertions internalize scope graph construction, whereas the queries internalize scope graph resolution. We discuss what a scope graph comprises, and present resolution in scope graphs as computing the answer to a *visibility query*.

A *scope graph* $\mathcal{G}$ is a triple $\langle S, E, \rho \rangle$ where S is a set of *node identifiers*, E is a multi-set of *labeled, directed edges*, and ρ is a *finite map* from node identifiers to terms. We will write $S_{\mathcal{G}}$, $E_{\mathcal{G}}$ and $\rho_{\mathcal{G}}$ for projecting the three components out of a graph $\mathcal{G}$, and may omit the subscript when it is unambiguous. We will refer to the term associated with a node identifier as the *datum* of a node. The complete syntax of graphs and terms is given in figure 6.9. We write ϵ for the empty graph and $\mathcal{G} \sqsubseteq \mathcal{G}'$ for the extension order on graphs. On sets we use the notation $X \cup Y$ and $X \sqcup Y$ to denote the union and the *disjoint* union of sets X and Y, respectively. We write $X \setminus Y$ for the set difference, and $x; X$ for $\{x\} \sqcup X$. We sometimes use set comprehension notation $\{\phi(x) \mid x \in X\}$ to denote the set of all elements $\phi(x)$ for x in X.

Scope graph: A triple $\mathcal{G} = \langle S, E, \rho \rangle$.

NAME RESOLUTION is modeled with *regular paths* in the graph. We write $\mathcal{G} \vdash p : s \xrightarrow{w} s_k$ to denote that p is a regular (acyclic) path in $\mathcal{G}$, starting in s, ending in s_k, and spelling the word w along its edges. We define the operations $\mathsf{src}\,(\_)$, $\mathsf{tgt}\,(\_)$ and $\mathsf{labels}\,(\_)$ to act on paths and project out the source node s, target node s_k, and list of labels w on the edges, respectively.

Regular paths: paths in the graph whose labeling matches a regular expression.

The answer to a *reachability query* $s \xrightarrow{r}\!\!\!\!\rightarrow D$ is a set of regular paths $s \xrightarrow{w} s'$ such that w matches the regular expression r and the datum of s' inhabits the term predicate D:

Reachability query: A reachability query $s \xrightarrow{r}\!\!\!\!\rightarrow D$.

$$\mathsf{Ans}\left(\mathcal{G}, s \xrightarrow{r}\!\!\!\!\rightarrow D\right) = \left\{ p \;\middle|\; \mathcal{G} \vdash p : s \xrightarrow{w} s' \text{ and } w \in \mathcal{L}(r) \text{ and } \rho_{\mathcal{G}}(s') \in D \right\}$$

We write $\mathcal{L}(r)$ for the set of words in the regular language described by r. A useful device when we consider partial reaching paths is the Brzozowski derivative [20] $\delta_w r$ of a regular expression r with respect to a word w, whose language is $\mathcal{L}(\delta_w r) = \{w' \mid ww' \in \mathcal{L}(r)\}$.

[20] Brzozowski 1964. "Derivatives of Regular Expressions"

Often we are interested in a refinement of reachability, which we call *visibility*. A datum is visible via a path p if and only if p is a least reaching path. Given reachability answer A, the subset of visible paths is defined as the minimum of A over a preorder R on paths:

Visibility: set of least reachable paths.

$$\mathsf{min}(A, R) = \{p \in A \mid \forall q \in A.\, Rqp \Rightarrow Rpq\}$$

Reachability is monotone with respect to graph extension: extending a graph with additional nodes and edges can only make *more* things reachable. In contrast, visibility is *non-monotonic* with respect to graph extension: extending a graph with additional nodes and edges may obscure—or, shadow—information that was previously visible.

We can now formally state the notion of stability of query answers that is key to the correct implementation of static name resolution: a query (answer) q is said to be *stable* between graphs $\mathcal{G} \sqsubseteq \mathcal{G}'$, when the answer set for the query is identical in both graphs: i.e. $\mathsf{Ans}\,(\mathcal{G}, q) = \mathsf{Ans}\,(\mathcal{G}', q)$.

Stable: A query answer is stable between graphs, if the answer is the same in both.

We now present the syntax of the constraint language Statix-core for making assertions about terms and an implicit, ambient scope graph. The syntax is generated over a signature:

Signature

l	$\in$	$\mathcal{I}$	label	x	$\in$	$\mathcal{X}$	term variable
f	$\in$	$\mathcal{F}$	symbol	z	$\in$	$\mathcal{Z}$	set variable
r	$\in$	$\mathcal{R}$	regex	s	$\in$	$\mathcal{V}$	node name

Syntax

Terms

$t \in \mathcal{T}$	::=	x	variable
	$\mid$	$f(t^*)$	compound term
	$\mid$	$l \mid s$	label and node

Sets of Terms

$\bar{t}$	::=	$z \mid \zeta$	set variable and set literal
ζ	::=	$\emptyset \mid \{t\}$	empty and singleton set
	$\mid$	$\zeta \sqcup \zeta$	disjoint union

Graphs

$\mathcal{G}$	::=	$\langle S \subseteq \mathcal{V}, \quad E \subseteq (\mathcal{V} \times \mathcal{I} \times \mathcal{V}), \quad \rho \subseteq (\mathcal{V} \rightharpoonup \mathcal{T}) \rangle$	

Constraints

C	::=	$\mathsf{emp} \mid \mathsf{false}$	true and false
	$\mid$	$C * C \mid t = t$	separating conjunction and term equality
	$\mid$	$\exists x.\ C$	existential variable quantification
	$\mid$	$\mathsf{single}(t, \bar{t}) \mid \mathsf{min}(\bar{t}, R, \bar{t})$	set singletons and minimum
	$\mid$	$\forall x \text{ in } \bar{t}.\ C$	universal quantification over sets
	$\mid$	$\nabla t \mapsto t \mid t \xrightarrow{l} t$	node and edge assertion
	$\mid$	$\mathsf{query}\ t \xrightarrow{r}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ C \mid \mathsf{dataOf}(t, t)$	graph query and data retrieval

Figure 6.9: Syntax of Statix-core.

Terms t are either variables x, compound terms $f(t^*)$, graph edge-labels l, graph nodes s, or graph edges $t \xrightarrow{l} t$. Importantly, nodes only appear as an artifact of substitution in the operational semantics and do not appear in source constraint problems. Literals for sets of terms $\bar{t}$ are used to represent query answer sets in programs and are generated from the disjoint union of singletons and empty sets. Sets of terms are implicitly understood to exist up to reordering.

Constraints C define assertions on terms and an underlying scope graph. As we will see subsequently, constraint satisfaction uses a notion of *ownership*, which gives the semantics a separation logic[21] flavor. This is reflected in the syntax of Statix-core where we use $C *$ ▲

[21] O'Hearn et al. 2001. "Local Reasoning about Programs that Alter Data Structures"

C for *separating* conjunction, and emp and false for the neutral and absorbing elements of $*$, respectively. The $t_1 = t_2$ constraint asserts that t_1 and t_2 are equal. The x binder in existential quantification $\exists x.\ C$ ranges over all possible terms, whereas the x in universal quantification $\forall x \text{ in } \bar{t}.\ C$ ranges over members in a given finite set of terms $\bar{t}$.

The assertions on the ambient scope graph $\mathcal{G}$ come in two flavors: node and edge assertion. The former is written $\nabla t_1 \mapsto t_2$ and asserts that t_1 is a node $s \in S_\mathcal{G}$ such that $\rho_\mathcal{G}(s) = t_2$. The node assertion gets unique ownership of s, such that no other node assertion can observe the same fact about the model $\mathcal{G}$. Similarly, edge assertions $t_1 \xrightarrow{l} t_2$ assert unique ownership of an edge $(t_1, l, t_2) \in E_\mathcal{G}$. The $\mathsf{dataOf}(t_1, t_2)$ constraint asserts that the data associated with node t_1 is t_2.

Query constraints $(\mathsf{query}\ t \xrightarrow{r}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ C)$ internalize reachability queries: we query node t for the set of all reaching paths over the regular expression r to nodes whose data satisfy the predicate D, and bind the query result to z in C. Queries yield *sets* of paths (embedded as terms) which motivates the need for set literals, universal quantification over the elements in the set, and the $\mathsf{single}(t, \bar{t})$ constraint which asserts that $\bar{t}$ is a singleton set containing just the element t. The constraint $\mathsf{min}(\bar{t}, R, \bar{t}')$ asserts that the latter set of terms is the minimum of the former over the preorder R and is used to specify disambiguation of a set of reaching paths to the set of visible paths. We implicitly convert between (finite) mathematical sets and term set syntax where necessary. We assume that the set $\mathcal{F}$ of term constructor symbols contains the necessary constructors to encode paths.

THE MEANING OF CONSTRAINTS is given by the inductively defined *constraint satisfaction* relation $\mathcal{G} \vDash_\sigma C$, which says that the graph $\mathcal{G}$ satisfies the closed constraint C with *graph support* $\sigma = \langle S, E \rangle$, where $S \subseteq S_\mathcal{G}$ and $E \subseteq E_\mathcal{G}$. In case the satisfaction judgment holds, we say that $\mathcal{G}$ is a *model* for the constraint C. Because the satisfaction relation does not explain how one finds a model, we also call this the *declarative semantics* of Statix-core.

Graph support: The part of the scope graph that is asserted by a constraint

$\mathcal{G} \vDash_\sigma C$ — Scope graph $\mathcal{G}$ satisfies constraint C with support σ

$$\textsc{Emp}\ \frac{}{\mathcal{G} \vDash_\bot \mathsf{emp}} \qquad \textsc{Conj}\ \frac{\mathcal{G} \vDash_{\sigma_1} C_1 \qquad \mathcal{G} \vDash_{\sigma_2} C_2}{\mathcal{G} \vDash_{\sigma_1 \sqcup \sigma_2} C_1 * C_2} \qquad \textsc{Eq}\ \frac{t_1 = t_2}{\mathcal{G} \vDash_\bot t_1 = t_2} \qquad \textsc{Exists}\ \frac{\mathcal{G} \vDash_\sigma C\,[t/x]}{\mathcal{G} \vDash_\sigma \exists x.\ C} \qquad \textsc{Singleton}\ \frac{}{\mathcal{G} \vDash_\bot \mathsf{single}(t, \{t\})}$$

$$\textsc{Min}\ \frac{\bar{t}' = \mathsf{min}(\bar{t}, R)}{\mathcal{G} \vDash_\bot \mathsf{min}(\bar{t}, R, \bar{t}')} \qquad \textsc{Forall-Empty}\ \frac{}{\mathcal{G} \vDash_\bot \forall x \ \mathsf{in}\ \emptyset.\ C} \qquad \textsc{Forall}\ \frac{\mathcal{G} \vDash_{\sigma_1} C\,[t_1/x] \qquad \mathcal{G} \vDash_{\sigma_2} \forall x \ \mathsf{in}\ \bar{t}_2.\ C}{\mathcal{G} \vDash_{\sigma_1 \sqcup \sigma_2} \forall x \ \mathsf{in}\ (\{t_1\} \sqcup \bar{t}_2).\ C} \qquad \textsc{Node}\ \frac{s \in S_\mathcal{G} \qquad \rho_\mathcal{G}(s) = t}{\mathcal{G} \vDash_{\langle s, \emptyset \rangle} \nabla s \mapsto t}$$

$$\textsc{Edge}\ \frac{(s_1, l, s_2) \in E_\mathcal{G}}{\mathcal{G} \vDash_{\langle \emptyset, (s_1, l, s_2) \rangle} s_1 \xrightarrow{l} s_2} \qquad \textsc{Query}\ \frac{\mathcal{G} \vDash_\sigma C\left[\mathsf{Ans}\left(\mathcal{G}, s \xrightarrow{r}\!\!\!\!\rightarrow D\right)/z\right]}{\mathcal{G} \vDash_\sigma \mathsf{query}\ s \xrightarrow{r}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ C} \qquad \textsc{Data}\ \frac{\rho_\mathcal{G}(s) = t}{\mathcal{G} \vDash_\bot \mathsf{dataOf}(s, t)}$$

Figure 6.10: Statix constraint satisfiability.

The first thing to be understood is the role of the graph support, which declaratively expresses *ownership* of graph structure by constraints. This ownership is distributed linearly: graph nodes and edges cannot be owned twice. This is clearly visible in the rule Conj for conjunctions $C_1 * C_2$, which disjointly distributes the support of the conjunction over the constraints C_1 and C_2.[22] Ownership is established by graph assertions (rules Node and Edge). Many primitive constraints (e.g., Eq and Data) have empty support ($\bot$).

We are only interested in models of constraints that are *supported*:

$$\frac{\mathcal{G} \vDash_{\langle S_\mathcal{G}, E_\mathcal{G} \rangle} C}{\mathcal{G} \vDash C}\ \textsc{Supported}$$

Intuitively, a model $\mathcal{G}$ is supported by a constraint C when every node and edge in it is asserted by C. For top-level constraints, we are exclusively interested in supported models. Models that are *not fully supported* at the top-level contain "junk": graph structure that is not asserted by the Statix specification. For our problem domain it does not make sense to consider those models, as they would contain binding structure that does not correspond to the input program.

Not every constraint that has a model also has a supported one. Consider, for example, the following constraint:

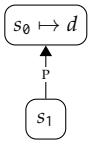
$$\exists s.\ \left(\mathsf{query}\ s \xrightarrow{P*}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ (\exists x.\ \mathsf{single}(x, z))\right)$$

The constraint existentially quantifies over an s, which is queried for the set of regular paths, so that:

$$z = \mathsf{Ans}\left(\mathcal{G}, s \xrightarrow{P^*}\!\!\!\!\rightarrow D\right)$$

▲

[22] We lift set operations pointwise to graph support. A particularly important operation is the disjoint union, written $\sigma_1 \sqcup \sigma_2$, which is defined as $\sigma_1 \cup \sigma_2$, if and only if $\sigma_1 \cap \sigma_2$ is empty.



▲

Figure 6.11: An *unsupported* model of the constraint on the left, assuming $d \in D$.

We then constrain the answer to be a singleton, containing only the existentially quantified path x. Whenever D is inhabited, there are clearly graphs that satisfy the constraint (e.g., figure 6.11). None of those graphs are supported, however, because there are no node or edge assertions in the constraint. This means that the whole constraint has empty support and the empty graph is hence not a supported model of the query. Because we are only interested in supported models, a Statix constraint solver should output that this constraint is unsatisfiable.

We use the declarative semantics to define the semantics of constraint entailment ($\Vdash$) and equivalence ($\dashv\Vdash$). This will enable us to reason about Statix-core specifications, so that we can prove, in particular, the desired properties of the operational semantics. We will need to reason about open constraints as well, which motivates us to first lift the declarative semantics to open constraints in the usual way. That is, we write $\mathcal{G}, \varphi \vDash_\sigma C$ to denote that the open constraint is satisfied by the model $\mathcal{G}$ when closed with the substitution φ ($\mathcal{G} \vDash_\sigma C\varphi$). Entailment and equivalence are then defined as follows:

Figure 6.12: Constraint entailment and equivalence.

▼

Entailment

$$\frac{\forall \mathcal{G}, \varphi, \sigma.\ (\mathcal{G}, \varphi \vDash_\sigma C_1 \text{ implies } \mathcal{G}, \varphi \vDash_\sigma C_2)}{C_1 \Vdash C_2}$$

Equivalence

$$\frac{C_1 \Vdash C_2 \qquad C_2 \Vdash C_1}{C_1 \dashv\Vdash C_2}$$

The role of support in constraint satisfiability gives the logic that we obtain for these entailment relations the flavor of a linear separation logic. That is, we get equivalence rules that one would expect for a linear separation logic. For example:

$$C_1 * C_2 \dashv\Vdash C_2 * C_1$$
$$C_1 * (C_2 * C_3) \dashv\Vdash (C_1 * C_2) * C_3$$
$$\mathsf{emp} * C \dashv\Vdash C$$
$$\mathsf{false} * C \dashv\Vdash \mathsf{false}$$

Conjunction is commutative and associative and has emp as its identity and false as the absorbing element. These hold because the equivalences linearly preserve support.

On the other hand, the following equivalences *fail* to hold, because the left and right hand side have different graph support:

$$C_1 * C_2 \dashv\vdash C_1 \quad \text{(does not hold)}$$
$$C_1 * C_2 \dashv\vdash C_2 \quad \text{(does not hold)}$$

6.3 Solving Constraints

Our goal is to derive, from the Statix specification of a type system, an executable type checker. A *sound* type checker should take a specification and an input program e and *construct* the ambient scope graph $\mathcal{G}$ such that $\mathcal{G}$ and e together obey the specification. Or, if and only if the program does not obey the specification, produce a name- or type-error.[23] Our approach to this is to equip Statix-core with an operational semantics that reduces constraints, as generated over a program, to a graph that satisfies the constraint according to the declarative semantics, or rejects the constraint *if and only if* such a graph does not exist. In this section we describe such an operational semantics *without queries*. We show that the operational semantics enjoys confluence and soundness with respect to the declarative semantics. In section 6.4 we extend this semantics to queries.

[23] This leaves room for *incompleteness*—i.e., type checker may get *stuck* and neither accept nor reject a program, but get into an infinite loop, or report that it could not make a decision.

The operational semantics of Statix without queries is a small-step semantics defined as a binary relation on state tuples $\langle \mathcal{G} \mid \overline{C} \rangle$, where $\mathcal{G}$ is a graph and $\overline{C}$ is a set of constraints that is repeatedly simplified. Semantically we treat the constraint set as a large conjunction and we non-deterministically pick a constraint from this set to perform a step on. The rules of the small-step semantics are displayed in figure 6.13.

A constraint C is solved by constructing an initial state κ as $\langle \epsilon \mid \{C\} \rangle$ and repeatedly stepping until a final or stuck state κ' is reached. We say that the operational semantics *accepts* C iff it reaches a final state $\langle \mathcal{G} \mid \emptyset \rangle$ and *rejects* C iff it reaches a final state $\langle \mathcal{G} \mid \{\mathsf{false}\} \rangle$. Any other states in which we cannot reduce by taking a step are said to be *stuck*.

The rules for the usual logical connectives (emp, false, $C_1 * C_2$, $=$, $\exists$, $\forall$, and single) are standard. The rule for set minimums proceeds by computation. For $\nabla t_1 \mapsto t_2$ there are two rules: if t_1 is a variable x, rule Op-Node-Fresh will extend the graph with a fresh node s, claim ownership over it, and substitute s for x everywhere.

$\kappa \to \kappa'$ — State κ steps to κ'

OP-EMP
$$\langle \mathcal{G} \mid \mathsf{emp}; \overline{C} \rangle \to \langle \mathcal{G} \mid \overline{C} \rangle$$

OP-FALSE
$$\langle \mathcal{G} \mid \mathsf{false}; \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle$$

OP-CONJ
$$\langle \mathcal{G} \mid (C_1 * C_2) ; \overline{C} \rangle \to \langle \mathcal{G} \mid C_1; C_2; \overline{C} \rangle$$

OP-EQ-TRUE
$$\frac{t_1\varphi = t_2\varphi \qquad \varphi \text{ is most general}}{\langle \mathcal{G} \mid (t_1 = t_2) ; \overline{C} \rangle \to \langle \mathcal{G}\varphi \mid \overline{C}\varphi \rangle}$$

OP-EQ-FALSE
$$\frac{\neg \exists \varphi. t_1\varphi = t_2\varphi}{\langle \mathcal{G} \mid (t_1 = t_2) ; \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

OP-EXISTS
$$\frac{y \text{ is fresh for } \mathcal{G} \text{ and } \overline{C}}{\langle \mathcal{G} \mid (\exists x.\ C) ; \overline{C} \rangle \to \langle \mathcal{G} \mid C\,[y/x] ; \overline{C} \rangle}$$

OP-SINGLETON-TRUE
$$\langle \mathcal{G} \mid \mathsf{single}(t, \{t'\}); \overline{C} \rangle \to \langle \mathcal{G} \mid (t = t'); \overline{C} \rangle$$

OP-SINGLETON-FALSE
$$\frac{\neg \exists t'. \bar{t} = \{t'\}}{\langle \mathcal{G} \mid \mathsf{single}(t, \bar{t}); \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

OP-MIN
$$\frac{\zeta' = \mathsf{min}(\zeta, R)}{\langle \mathcal{G} \mid \mathsf{min}(\zeta, R, x); \overline{C} \rangle \to \langle \mathcal{G}\,[\zeta'/x] \mid \overline{C}\,[\zeta'/x] \rangle}$$

OP-FORALL
$$\langle \mathcal{G} \mid (\forall x \text{ in } \zeta.\ C) ; \overline{C} \rangle \to \langle \mathcal{G} \mid \{C\,[t/x] \mid t \in \zeta\} \sqcup \overline{C} \rangle$$

OP-NODE-FRESH
$$\frac{s \notin S}{\langle \langle S, E, \rho \rangle \mid (\nabla x \mapsto t); \overline{C} \rangle \to \langle \langle (s; S), E, \rho[s \to t]\,[s/x] \rangle \mid \overline{C}\,[s/x] \rangle}$$

OP-NODE-STALE
$$\frac{t_2 \text{ is not a variable}}{\langle \mathcal{G} \mid (\nabla t_2 \mapsto t_1) ; \overline{C} \rangle \to \langle \mathcal{G} \mid \{\mathsf{false}\} \rangle}$$

OP-DATA
$$\frac{\rho(s) = t_2}{\langle \mathcal{G} \mid \mathsf{dataOf}(s, t_1); \overline{C} \rangle \to \langle \mathcal{G} \mid (t_1 = t_2); \overline{C} \rangle}$$

OP-EDGE
$$\langle \langle S, E, \rho \rangle \mid (s_1 \xrightarrow{l} s_2); \overline{C} \rangle \to \langle \langle S, (s_1, l, s_2) ; E, \rho \rangle \mid \overline{C} \rangle$$

Figure 6.13: Operational semantics of Statix without queries.

If t_1 is not a variable, specifically if it is a node, then it must be owned already and the rule OP-NODE-STALE rejects the constraint by stepping to $\{\mathsf{false}\}$. For example, both rules would be executed once for the specification $\nabla x \mapsto () * \nabla x \mapsto ()$: one of the constraints gets ownership, and the other fails to get it. Edge assertions $t_1 \xrightarrow{l} t_2$ construct new edges in the graph via OP-EDGE when both endpoints have become nodes. Multiple edges with the same label between the same endpoints can exist separately—i.e., there is no edge counterpart to the be OP-NODE-STALE rule. Data assertions $\mathsf{dataOf}(t_1, t_2)$ compute by unification when the node t_1 becomes ground.

WE WILL SHOW that the operational interpretation of a Statix-core specification is sound with respect to the declarative reading. That

▲

is, if the operational semantics accepts a constraint C, then the resulting graph is a *supported model* for C. And additionally, if the operational semantics rejects a constraint C, then there exists no supported model for C. From the perspective of the object language semantics defined in Statix-core this means that the derived type-checker is sound by construction with respect to the typing rules of the language.

If we extend our declarative semantics for constraints to states, we can state the soundness criterion more concisely and uniformly. We accomplish this via an *embedding* of states into constraints:

Definition 1. *The* embedding of a graph $\langle V, E, \rho \rangle$ *and the* embedding of a state $\langle \mathcal{G} \mid \overline{C} \rangle$ *are defined as follows:*

$$[\![\langle V, E, \rho \rangle]\!] = \left(\mathop{\Large *}_{s \in V} \nabla s \mapsto \rho(s) \right) * \left(\mathop{\Large *}_{(s,l,s') \in E} \left(s \xrightarrow{l} s' \right) \right)$$

$$[\![\langle \mathcal{G} \mid \overline{C} \rangle]\!] = [\![\mathcal{G}]\!] * (\mathop{\Large *} \overline{C})$$

The soundness criterion can now be stated in terms of constraint equivalence between initial and final states. Specifically, we will show that the following theorem holds:

Theorem 1 (Soundness of Statix-core without queries)**.** *Let κ be either an accepting or rejecting state. The operational semantics for Statix-core without queries is* sound*:*

$$\langle \epsilon \mid \{C\} \rangle \rightarrow^* \kappa \;\; \textit{implies} \;\; C \dashv\vdash [\![\kappa]\!]$$

This is equivalent to the aforementioned informal definition of soundness, which can be shown using the facts that top-level constraints are closed and that graphs are trivially a model for their own embedding. We would like to prove this statement by induction on the trace of steps. This requires us to show that individual steps operate along constraint equivalences—i.e., that $\kappa_1 \rightarrow \kappa_2$ implies $[\![\kappa_1]\!] \dashv\vdash [\![\kappa_2]\!]$. Indeed, this is the case for many of the rules. For example, OP-CONJ and OP-EMP rewrite along commutativity, associativity, and identity of the separating conjunction. The rules for existential quantification and node assertion, however, cannot be justified using logical equivalences. To this end we define a more general notion of *preserving satisfiability*:

Definition 2. *We write $C_1 \mathrel{|\!\sim} C_2$ to denote that C_2 is satisfiable when C_1 is satisfiable, that is, the existence of a model $\mathcal{G}$ for open constraint C_1,*

implies that $\mathcal{G}$ is also a model for C_2, but modulo graph equivalence ($\approx$):

$$\frac{\forall \mathcal{G}_1, \varphi_1.\ (\mathcal{G}_1, \varphi_1 \vDash C_1 \textit{ implies } (\exists \mathcal{G}_2, \varphi_2.\ \mathcal{G}_2, \varphi_2 \vDash C_2 \quad \textit{s.t. } \mathcal{G}_1 \approx \mathcal{G}_2))}{C_1 \mid\sim C_2}$$

We also define the symmetric counterpart, which denotes equi-satisfiability:

$$\frac{C_1 \mid\sim C_2 \wedge C_1 \sim\mid C_2}{C_1 \sim\mid\sim C_2}$$

For top-level (closed) constraints this notion of preserving satisfiability coincides with constraint equivalence. Furthermore, constraint entailment $C_1 \Vdash C_2$ always implies $C_1 \mid\sim C_2$, allowing the use of laws such as identity, commutativity, and associativity of the separating conjunction when we reason about preservation of satisfiability. Steps in the operational semantics are semantically justified in that they preserve satisfiability of the constraint problem:

Lemma 2. *Steps preserve satisfiability: $\kappa_1 \to \kappa_2$ implies $[\![\kappa_1]\!] \sim\mid\sim [\![\kappa_2]\!]$*

This may feel counter-intuitive, as steps construct a graph and preservation of satisfiability demands equivalent graphs as the model for the left- and right-hand-sides of the step. The key to understanding this lies in definition 1 of the state embedding together with the rules for graph construction Op-Node-True and Op-Edge, which show that bits of graph (support) are *merely moved* between the constraint program and the (partial) model. In the initial state the entire model should be specified in the input constraint and in the final state the entire model is a given.

Proof sketch. The proof is by case analysis on the constraint that is the focus of the step. Many cases can indeed be proven using logical equivalences. Other cases, such as the elimination of existential quantifiers rely on the commutativity of substitutions with embedding of states. The graph equivalence is trivial everywhere, except for the step Op-Node-True. An arbitrary *fresh* node is chosen there, which means that the models for the different sides of the step are equal only up-to renaming of nodes. □

As a consequence of lemma 2, the operational semantics enjoys soundness with respect to the declarative semantics (theorem 1):

Proof sketch for theorem 1. The embeddings of the initial and final states reduce to C and $[\![\mathcal{G}]\!]$ respectively. We repeatedly apply the fact that steps preserve satisfiability and prove $C \sim\!|\!\sim [\![\mathcal{G}]\!]$. Now we make use of the fact that graphs are trivially a supported model for their own embedding: $\mathcal{G} \vDash [\![\mathcal{G}]\!]$. By the above constraint equivalence, $\mathcal{G}$ must then also be a supported model for C up-to renaming of nodes. The theorem follows from the fact that constraint satisfaction is preserved by consistent renaming of nodes in the model and the constraint, and the fact that node renaming vanishes on top-level constraints. □

The operational semantics is non-deterministic, but confluent. This can be shown to hold by proving the diamond property for the reflexive closure of the step relation. We give a proof outline for confluence using the diamond property at the end of section 6.4.

Theorem 3 (Confluence)**.** *If $\kappa \to^* \kappa_1$ and $\kappa \to^* \kappa_2$ then there exists κ_1' and κ_2' such that $\kappa_1 \to^* \kappa_1'$ and $\kappa_2 \to^* \kappa_2'$ where $\kappa_1' \approx \kappa_2'$.*

6.4 Solving Queries: Knowing When to Ask

We now address the problem of extending the Statix-core operational semantics to support queries. First we improve our understanding of the problem, by considering a naive semantics that answers queries unconditionally. We show that this approach yields unsound name resolution by violating answer stability. A rule for queries needs to ensure that query answers are stable. We develop the sound rule in three steps:

- We characterize the scope graph extensions that causes query answer instability and show that we can guarantee stability by ensuring the *absence of (weakly) critical edge* extensions.
- We describe a fragment of *well-formed* constraint programs for which it is feasible to check, without the full power of constraint solving, that certain graph edges cannot exist in any future graph (figure 6.17), addressing the problem that the complete scope graph is unknown during type checking.
- We obtain an operational semantics for well-formed Statix-core constraints *with queries* by *guarding* query simplification by the absence of weakly critical edges in all future graphs. We prove

that this guarded rule preserves satisfiability and thus yields a sound, non-backtracking operational semantics (theorem 11).

CONSIDER A NAIVE, unconditional evaluation rule for queries:

OP-QUERY-NAIVE

$$\frac{\overline{t} = \mathsf{Ans}\left(\mathcal{G}, s \xrightarrow{r}\!\!\!\!\rightarrow D\right)}{\left\langle \mathcal{G} \mid \left(\mathsf{query}\ s \xrightarrow{r}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ C\right); \overline{C}\right\rangle \to \left\langle \mathcal{G} \mid C\left[\overline{t}/z\right]; \overline{C}\right\rangle}$$

It solves a query by answering the given reachability query in the incomplete graph that is part of the solver state at that time. It then simplifies the constraint program by substituting the answer set into C. This rule is *unsound*: it results in graphs that are not models of the input constraint.

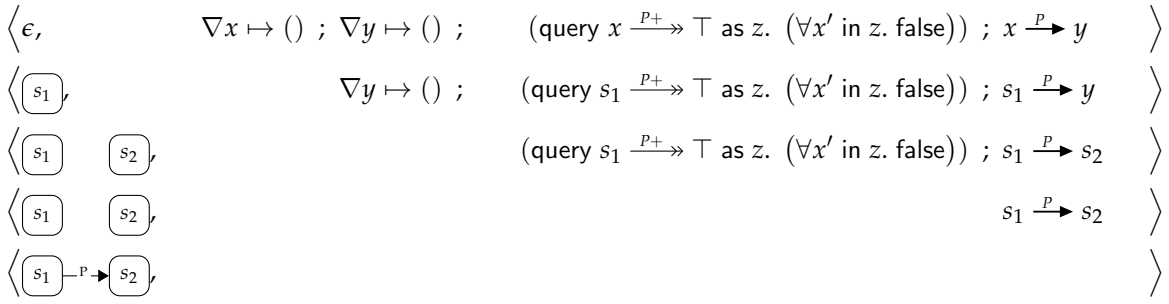
To understand how this rule is unsound, consider the following example constraint set $\overline{C}$:

$$\begin{aligned}\overline{C} = {} & \nabla x \mapsto () \\ & ; \nabla y \mapsto () \\ & ; \mathsf{query}\ x \xrightarrow{P+}\!\!\!\!\rightarrow \top\ \mathsf{as}\ z.\ \left(\forall x'\ \mathsf{in}\ z.\ \mathsf{false}\right) \\ & ; x \xrightarrow{P} y\end{aligned}$$

The query in these constraints asks for any node that is reachable in the graph after traversing at least one P-labeled edge, starting in the node for the variable x. It then asserts (via $\forall x'$ in z. false) that the answer to this query is empty. We can trace the evaluation of this example:

Figure 6.14: Trace demonstrating unsoundness of a naive query simplification rule.

▼



The final graph in figure 6.14 is *not* a model for the input constraint. The answer to the query in the final graph is non-empty: there

is a single path in the answer consisting of the only edge in the graph. The reason for this faulty behavior can be reduced to two observations: (1) the naive solver answers queries based on incomplete information, namely the partial graph that happens to be part of its state at that point in the trace, and (2) query answers are in general not stable under graph extensions that occur later in the constraint solver. This raises the question: what additional conditions must hold in a given state such that query solving *is sound*—i.e., under what side-condition is the following rule for query answering sound?

$$\langle \mathcal{G} \mid \left(\mathsf{query}\ s \xrightarrow{r}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ C\right); \overline{C} \rangle \quad \rightarrow \quad \langle \mathcal{G} \mid C\left[\mathsf{Ans}\left(\mathcal{G}, s \xrightarrow{r}\!\!\!\!\rightarrow D\right)/z\right]; \overline{C} \rangle$$

In order to prove that this rule is sound, it suffices to prove that it preserves satisfiability, as is the case for the other steps of the operational semantics (c.f. lemma 2). Concretely, to show that this rule preserves satisfiability, we have to prove:

$$[\![\mathcal{G}]\!] * \left(\mathsf{query}\ s \xrightarrow{r}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ C\right) * (\ast\overline{C}) \quad \sim\!\!|\!\sim \quad [\![\mathcal{G}]\!] * C\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right)/z\right] * (\ast\overline{C})$$

This means that every supported model $\mathcal{G}'$ for the left constraint must be a supported model for the right constraint as well, and vice versa. When is this the case? It holds exactly when the query $s \xrightarrow{r}\!\!\!\!\rightarrow D$ is stable for the graph extension $\mathcal{G} \sqsubseteq \mathcal{G}'$. Or, in the terms of the application domain of Statix, it holds if all *relevant* namebinding information that may influence resolution of the specified name is present in $\mathcal{G}$. That means, for example, that no further names will be discovered in the remainder of the program that shadow declarations that are reachable in the current graph $\mathcal{G}$.

Ensuring Answer Stability. To guarantee query stability, we want to prevent the solver from extending the graph with critical edges. We argue however that the absence of critical edges is too strong a notion for a solver to verify. To remedy this, we derive the notion of a *weakly critical edge* which only considers the extension boundary.

To appoint a *root cause of instability* of reachability queries under graph extensions $\mathcal{G} \sqsubseteq \mathcal{G}'$, we focus on paths that exist in $\mathcal{G}'$, but not in $\mathcal{G}$:

$$p \in \mathsf{Ans}\left(\mathcal{G}', s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right) \setminus \mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right)$$

Because the start node of every path in the answer set of a query is fixed (in this case to s_1) they can always be partitioned into a non-empty prefix in $\mathcal{G}$ and the remainder. The first edge of the remainder can be considered the root cause for this new path in $\mathcal{G}'$. We call such edges *critical*.

Definition 3. *An edge $(s_1, l, s_2) \in E_{\mathcal{G}'}$ is called* critical *with respect to a graph extension $\mathcal{G} \sqsubseteq \mathcal{G}'$ and a query $s \xrightarrow{r}\!\!\!\!\rightarrow D$ if there exist paths p_1 and p_2 that satisfy the following conditions:*

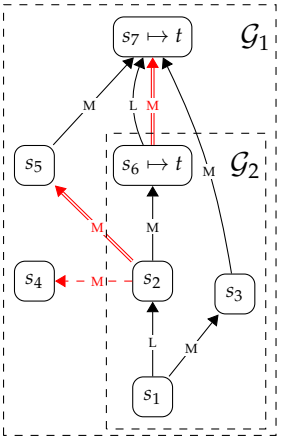
(a) $\mathcal{G} \vdash p_1 : s \xrightarrow{w_1} s_1$ *for some word* w_1,

(b) $\mathcal{G}' \vdash p_2 : s_2 \xrightarrow{w_2} s_3$ *for some node* s_3 *and word* w_2,

(c) $(p_1 \cdot l \cdot p_2) \in \mathsf{Ans}\left(\mathcal{G}', s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right)$, *and*

(d) $(s_1, l, s_2) \notin E_{\mathcal{G}}$.

Figure 6.15 visualizes the critical edges for a particular graph extension and query. Critical edges for a query are interesting because *their absence* in a graph extension guarantees stability of the answer to that query:



◀

Figure 6.15: Critical edges for the query $s_1 \xrightarrow{LM^*}\!\!\!\!\rightarrow D$, assuming that $t \in D$. Double solid red arrows depict critical edges, whereas dashed red arrows depict weakly critical edges. Every critical edge is also weakly critical.

Lemma 4 (Absence of Critical Edges)**.** *A reachability query $s \xrightarrow{r}\!\!\!\!\rightarrow D$ is stable under graph extension $\mathcal{G} \sqsubseteq \mathcal{G}'$ iff $\mathcal{G} \sqsubseteq \mathcal{G}'$ contains no critical edges for $s \xrightarrow{r}\!\!\!\!\rightarrow D$.*

Proof. The absence of critical edges implies stability because *every* path that answers a query that is in the extended graph $\mathcal{G}'$ *but not* in the original graph $\mathcal{G}$ can be partitioned as $p_1 \cdot l \cdot p_2$ such that

$(\mathsf{tgt}(p_1), l, \mathsf{src}(p_2))$ is a critical edge. Consequently, the absence of critical edges in an extension immediately implies that the extended graph yields no new answers to the query under scrutiny. The other direction of this lemma holds trivially. □

As indicated by lemma 4, it would be sufficient for the rule for queries to require the absence of critical edges in future graphs. However, the problematic question is: critical with respect to which graph extension? Indeed, the graphs $\mathcal{G}'$ that lemma 4 quantifies over are *all future graphs* of a trace in the operational semantics. Precisely knowing $\mathcal{G}'$ is as difficult as solving the constraint program. Hence it is not feasible for a solver to guard against the absence of critical edges with pinpoint accuracy. In the remainder of this section we describe a two-part approach to sound operation of a non-backtracking solver based on *over-approximating* the criticality of an edge.

BECAUSE THE NOTION of a critical edge is derived from entire new reaching paths in graph extensions, guarding against critical edge extensions requires looking ahead over arbitrary constraint solving. Our approximation, a *weakly critical* edge, reduces the required lookahead to just one-edge extensions of the *current* graph:

Definition 4. *An edge (s_1, l, s_2) is called* weakly critical *with respect to a graph $\mathcal{G}$ and a query $s \xrightarrow{r}\!\!\!\!\rightarrow D$ if there exists a path p_1 that satisfies the following conditions:*

(a) $\mathcal{G} \vdash p_1 : s \xrightarrow{w_1} s_1$ *for some word w_1,*

(b) the word $(w_1 l)$ is a prefix of some word in $\mathcal{L}(r)$, and

(c) $(s_1, l, s_2) \notin E_{\mathcal{G}}$.

In figure 6.15 an edge is highlighted that is only weakly critical: it shares all the features of a critical edge except that it does not actually give rise to new paths in the answer set of the query. The intuition behind a weakly critical edge is that it *may* lead to additional reaching paths. Every critical edge is also weakly critical, so that the following corollary holds:

Corollary 5. *A reachability query $Q = s \xrightarrow{r}\!\!\!\!\rightarrow D$ is stable under graph extension $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ if (but not* iff*) the graph extension $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ contains no edges that are weakly critical for Q.*

Proof. Every critical edge is also weakly critical because $(p_1 \cdot l \cdot p_2) \in \mathsf{Ans}\left(\mathcal{G}', s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right)$ implies that (wl) is a prefix of some word in $\mathcal{L}(r)$, for $w = \mathsf{labels}\,(p_1)$. The conclusion then immediately follows from lemma 4. □

Because visibility is defined as the minimum of a reachability query answer (section 6.2), the absence of weakly critical edges is also a *sufficient* (but not necessary) condition for stability of visibility query answers.

Corollary 6 (Absence of Weakly Critical Edges)**.** *A visibility query $Q = s \xrightarrow{r}\!\!\!\!\rightarrow D$ is stable under graph extension $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ if the graph extension $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ contains no edges that are weakly critical for Q.*

Consequently, the absence of weakly critical edges is also sufficient to guarantee the soundness of visibility queries with *any* path order $\leq_{\mathsf{p}}$. However, for particular choices of the path order there exist tractable approximations of criticality of edges for stability of reachability that are more precise than weak criticality. For example, the path ordering is often defined as the lexicographical extension of a precedence ordering on edge labels. Edge extensions of the graph with lower precedence than existing edges can in that case be disregarded as influential to name resolution. Our results extend to such refinements in a straightforward manner.

By means of a well-formedness judgment $\vdash C$ wf on Statix-core constraints, we define a large class of constraints for which we can check the absence of weakly critical edges. To this end we will also define a predicate $C \not\rightsquigarrow (s, l)$ which can be checked syntactically, but has the semantics that C does not support any l-edges out of s *if* C is well-formed. We then prove the following *guarded query simplification rule* correct:

Figure 6.16: An evaluation step for queries guarded by a condition that prohibits answering it if other constraints in C may support critical edges.

▼

Op-Query-Guarded

$$\frac{\forall s_2, l.\, \left(\mathcal{G} \vdash p : s_1 \xrightarrow{w} s_2 \text{ and } \mathcal{L}(\delta_{wl} r) \neq \emptyset \text{ implies } (C; \overline{C}) \not\rightsquigarrow (s_2, l)\right)}{\left\langle \mathcal{G} \mid \left(\mathsf{query}\ s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ C\right); \overline{C} \right\rangle \rightarrow \left\langle \mathcal{G} \mid C\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right)/z\right]; \overline{C} \right\rangle}$$

Recall that the condition $\mathcal{L}(\delta_{wl} r) \neq \emptyset$ means that (wl) is a prefix of some word in $\mathcal{L}(r)$. Intuitively, the precondition states that the remainder of the constraint program does not support any weakly critical edges for the query under scrutiny.

We define well-formedness inductively using the rules in figure 6.17. The intuition behind well-formed constraints is that asserting new outgoing edges on nodes requires *permission to extend* that scope. Permission is granted by a freshness assertion. The well-formedness judgment is defined in terms of an auxiliary judgment $\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C$ which denotes that the constraint C requires permission for variables in $\Delta^{\downarrow}$, and grants permission for those in $\Delta^{\uparrow}$.

$\vdash C$ wf — Constraint program C has sufficient permissions

$$\frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \Delta^{\downarrow} \subseteq \Delta^{\uparrow}}{\vdash C \text{ wf}} \textsc{Wf-Program}$$

$\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C$ — C requires permissions for variables in $\Delta^{\downarrow}$ and grants them for $\Delta^{\uparrow}$

$$\frac{}{\emptyset, \emptyset \vdash \text{emp}} \textsc{Wf-True} \qquad \frac{}{\emptyset, \emptyset \vdash \text{false}} \textsc{Wf-False}$$

$$\frac{\Delta_1^{\downarrow}, \Delta_1^{\uparrow} \vdash C_1 \qquad \Delta_2^{\downarrow}, \Delta_2^{\uparrow} \vdash C_2}{\Delta_1^{\downarrow} \cup \Delta_2^{\downarrow}, \Delta_1^{\uparrow} \cup \Delta_2^{\uparrow} \vdash C_1 * C_2} \textsc{Wf-Conj} \qquad \frac{}{\emptyset, \emptyset \vdash t_1 = t_2} \textsc{Wf-Eq}$$

$$\frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \left(x \in \Delta^{\downarrow} \Rightarrow x \in \Delta^{\uparrow}\right)}{\Delta^{\downarrow} \setminus \{x\}, \Delta^{\uparrow} \setminus \{x\} \vdash \exists x.\ C} \textsc{Wf-Exists} \qquad \frac{}{\emptyset, \emptyset \vdash \text{single}(t, \bar{t})} \textsc{Wf-Singleton}$$

$$\frac{}{\emptyset, \{x\} \vdash \nabla x \mapsto t} \textsc{Wf-Node-Var} \qquad \frac{t \text{ is not a variable}}{\emptyset, \emptyset \vdash \nabla t \mapsto t'} \textsc{Wf-Node-NoVar}$$

$$\frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \left(x \in \Delta^{\downarrow} \Rightarrow x \in \Delta^{\uparrow}\right)}{\Delta^{\downarrow} \setminus \Delta^{\uparrow} \setminus \{x\}, \emptyset \vdash \forall x \text{ in } \bar{t}.\ C} \textsc{Wf-Forall} \qquad \frac{}{\{x\}, \emptyset \vdash x \xrightarrow{l} t'} \textsc{Wf-Edge-Var}$$

$$\frac{t \text{ is not a variable}}{\emptyset, \emptyset \vdash t \xrightarrow{l} t'} \textsc{Wf-Edge-NoVar} \qquad \frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C}{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash \text{query } s \xrightarrow{r}\!\!\!\!\rightarrow D \text{ as } z.\ C} \textsc{Wf-Query}$$

$$\frac{}{\emptyset, \emptyset \vdash \text{dataOf}(t, t')} \textsc{Wf-Data}$$

Figure 6.17: Constraint well-formedness using an auxiliary permission relation.

The relation is easily extended to constraint sets, following the intuition that constraint sets are conjunctions.

Let us highlight some of the rules as an explanation of what the relation means. The rules Wf-Program asserts that a constraint is well-formed if it has sufficient permissions for the scope extensions that it supports. Many of the constraints neither require (↓), nor grant (↑) any permissions. For example, Wf-True, Wf-False and Wf-Eq each have empty requirements $\Delta^{\downarrow} = \emptyset$ and an empty set of granted permissions $\Delta^{\uparrow} = \emptyset$. The rule for Wf-Conj just combines permissions of the two operands of the separating conjunction.

▲

Permission is *granted* by the rule WF-NODE-VAR on a variable x by a node assertion $\nabla x \mapsto t$. Permission is *required* by the rule WF-EDGE-VAR on a variable x for the constraint $x \xrightarrow{l} t$. Rules with binders take care that the bound variable x has sufficient permission using a premise $x \in \Delta^{\downarrow} \Rightarrow x \in \Delta^{\uparrow}$.

We can relate permissions granted to the support of the constraint in any model.

Lemma 7. *A granted permission for a variable x in a constraint C guarantees that x stands for a supported scope in any model of C:*

$$\frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \mathcal{G}, \varphi \vDash_{\sigma} C \qquad x \in \Delta^{\uparrow}}{x\varphi = s \text{ and } s \in \sigma \text{ for some } s}$$

Proof sketch. The proof of this property is by induction over the permission derivation. Most cases are simple and follow from the fact that most constraints have an empty $\Delta^{\uparrow}$. The cases for the quantifiers and the conjunction are a matter of bookkeeping and invoking the induction hypothesis. In case of the rule WF-NODE-VAR we have $C = \nabla x \mapsto t$ for some x and the conclusion follows from inversion on the derivation of $\mathcal{G}, \varphi \vDash_{\sigma} \nabla x \mapsto t$, which proves that $x\varphi = s$ and that $\sigma = \langle s, \emptyset \rangle$. □

Syntactical extends judgment. The key observation is that this lemma makes it possible to ensure that a well-formed constraint set contains no weakly critical edges for a given scope s by simply scanning the constraint set for edge assertions with known source node s. We now define a simple judgment $C \hookrightarrow (s, l)$ that formalizes the observation that a constraint C certainly asserts an edge $(s, l, ...)$ and then prove that its negation $C \not\hookrightarrow (s, l)$ is sufficient evidence for the absence of weakly critical edges on s, provided that C is well-formed.

Figure 6.18: Syntactical edge-support relation.

▼

$C \hookrightarrow (s, l)$ — Constraint program C asserts an l-edge on known node s

EXT-CONJ1
$$\frac{C_1 \hookrightarrow (s, l)}{(C_1 * C_2) \hookrightarrow (s, l)}$$

EXT-CONJ2
$$\frac{C_2 \hookrightarrow (s, l)}{(C_1 * C_2) \hookrightarrow (s, l)}$$

EXT-EXIST
$$\frac{C \hookrightarrow (s, l)}{(\exists x.\, C) \hookrightarrow (s, l)}$$

EXT-EDGE
$$\frac{}{(s \xrightarrow{l} t) \hookrightarrow (s, l)}$$

EXT-FORALL
$$\frac{C \hookrightarrow (s, l)}{(\forall \overline{t} \text{ in } z.\, C) \hookrightarrow (s, l)}$$

EXT-QUERY
$$\frac{C \hookrightarrow (s, l)}{(\text{query } t \xrightarrow{r} D \text{ as } z.\, C) \hookrightarrow (s, l)}$$

The key lemma is as follows:

Lemma 8. *For all well-formed constraints the syntactical approximation of absence of support implies the semantic counterpart. That is:*

$$\frac{\vdash C \text{ wf} \qquad C \not\rightsquigarrow (s,l) \qquad \mathcal{G}, \varphi \vDash_\sigma C \qquad s \notin \sigma}{\forall s'.\,(s,l,s') \notin \sigma}$$

Proof sketch. We prove a stronger property:

$$\frac{\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C \qquad \Big(\forall (x \in \Delta^{\downarrow}) \Rightarrow (x\varphi \neq s)\Big) \qquad C \not\rightsquigarrow (s,l) \qquad \mathcal{G}, \varphi \vDash_\sigma C \qquad s \notin \sigma}{\forall s'.(s,l,s') \notin \sigma}$$

From the premise $\vdash C$ wf of lemma 8 the premise $\Delta^{\downarrow}, \Delta^{\uparrow} \vdash C$ follows by definition of well-formedness. The second premise $\forall (x \in \Delta^{\downarrow}) \Rightarrow (x\varphi \neq s)$ follows also from well-formedness of C. First, we observe that $x \in \Delta^{\downarrow}$ implies that $x \in \Delta^{\uparrow}$. Then we apply lemma 7 and prove that $x\varphi \in \sigma$. Because we know that $s \notin \sigma$, the conclusion $x\varphi \neq s$ follows.

The proof itself is by induction on C. The interesting case to consider is edge assertions. In case the source of the edge is ground, the conclusion follows from inversion of the third premise $(s' \xrightarrow{l'} t) \not\rightsquigarrow (s,l)$. In case the source of the edge is represented by a variable x, the first premise guarantees that $x \in \Delta^{\downarrow}$, such that the conclusion follows by the second premise. □

To use well-formedness like this—i.e., to guard against weakly critical edges in the constraint set—it is equally important that well-formedness is preserved by evaluation steps. That allows it to be checked only once on the input program without dynamically checking it on intermediate constraint sets.

Theorem 9. *Steps preserve well-formedness of constraints:*

$$\big(\langle \mathcal{G} \mid \overline{C}_1 \rangle \rightarrow \langle \mathcal{G}' \mid \overline{C}_2 \rangle \textit{ and } \vdash \overline{C}_1 \text{ wf}\big) \textit{ imply } \vdash \overline{C}_2 \text{ wf}$$

Proof sketch. There are a number of evaluation steps that influence permissions.

Any step that performs a substitutes in the constraint set can have impact on the required and granted permissions. We can show, however, that all substitutions preserve well-formedness, because their affect on granted and required permissions is symmetric.

The interesting rules are those for freshness assertions, which altogether remove a freshness assertion from the constraint set,

dropping a permission on a variable x. We can prove that the rule OP-NODE-FRESH preserves well-formedness by observing that the substitution of a new node for the variable x eliminates any required permissions on that variable. The rule OP-NODE-STALE preserves well-formedness, because the constraint set $\{\mathsf{false}\}$ is well-formed. □

These results enable us to prove the main new result of this section:

Theorem 10. *The guarded simplification step preserves satisfiability:*

$$\frac{\left(\forall s_2, l. \mathcal{G} \vdash p : s_1 \xrightarrow{w} s_2 \textit{ and } \mathcal{L}(\delta_{wl} r) \neq \emptyset \textit{ imply } (C; \overline{C}) \not\rightsquigarrow (s_2, l)\right)}{[\![\mathcal{G}]\!] * \left(\mathsf{query}\ s_1 \xrightarrow{r} \!\!\!\!\rightarrow D\ \mathsf{as}\ z.\ C\right) * (\divideontimes \overline{C}) \rightsquigarrow\!\!\!\!\mid\!\sim [\![\mathcal{G}]\!] * C\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r} \!\!\!\!\rightarrow D\right)/z\right] * (\divideontimes \overline{C})}$$

Proof sketch. We prove this equi-satisfiability result in the direction right to left. (The other direction proceeds similarly. By the definition of equi-satisfiability we assume a graph $\mathcal{G}'$ that is a supported model for the right-hand side:

$$\mathcal{G}', \varphi \vDash [\![\mathcal{G}]\!] * C\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r} \!\!\!\!\rightarrow D\right)/z\right] * (\divideontimes \overline{C}) \qquad \text{(I)}$$

To prove that $\mathcal{G}'$ is also a model for the left-hand side, we have to prove that the substituted answer to the query is stable for the extension $\mathcal{G} \sqsubseteq \mathcal{G}'$. The conjunction distributes support in disjoint fashion over the operands, and the embedding of $\mathcal{G}$ requires support for all of its nodes and edges. Consequently:

$$\mathcal{G}', \varphi \vDash_{\langle S_{\mathcal{G}'} \setminus S_{\mathcal{G}}, E_{\mathcal{G}'} \setminus E_{\mathcal{G}} \rangle} C\left[\mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r} \!\!\!\!\rightarrow D\right)/z\right] * (\divideontimes \overline{C}) \qquad \text{(II)}$$

Now assume a weakly critical edge (s_2, l, s_3). By definition we must have that $\mathcal{G} \vdash p : s_1 \xrightarrow{w} s_2$ and $\mathcal{L}(\delta_{wl} r) \neq \emptyset$. From the guard of the query simplification rule we may conclude $(C; \overline{C}) \not\rightsquigarrow (s_2, l)$. This relation is preserved under the answer set substitution into the constraint C. Lemma 8 now ensures that the remainder of the constraint program cannot support the weakly critical edge:

$$\forall s_3. (s_2, l, s_3) \notin (E_{\mathcal{G}'} \setminus E_{\mathcal{G}})$$

It follows by corollary 5 that the answer set is stable for this graph extension:

$$\mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r} \!\!\!\!\rightarrow D\right) = \mathsf{Ans}\left(\mathcal{G}', s_1 \xrightarrow{r} \!\!\!\!\rightarrow D\right) \qquad \text{(III)}$$

Combining (I) and (III), we have:

$$\mathcal{G}', \varphi \vDash [\![\mathcal{G}]\!] * C\left[\mathsf{Ans}\left(\mathcal{G}', s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right)/z\right] * \left(\divideontimes \overline{C}\right)$$

The desired result follows from introducing the query in the middle operand according to the satisfiability rule QUERY. □

We have proven that all steps in the extended operational semantics preserve satisfiability. Soundness follows:

Theorem 11 (Soundness of Statix-Core with Queries)**.** *If the operational semantics accepts a closed and* well-formed *constraint C, i.e. $\langle \epsilon \mid \{C\} \rangle \rightarrow^* \langle \mathcal{G} \mid \emptyset \rangle$, then the resulting graph is a supported model for that constraint: $\mathcal{G} \vDash C$. If C is rejected, then no supported model exists—i.e., there is no graph $\mathcal{G}$ such that $\mathcal{G} \vDash C$.*

Proof. The proof is the same as the proof for soundness of the fragment without queries, using theorem 10 to prove that the additional step in the operational semantics also preserves satisfiability. □

We end our discussion of the extended operational semantics by observing that it is still confluent. We prove it by proving the following diamond property:

Lemma 12 (Diamond Property for Reflexive Closure)**.** *If $\kappa \rightarrow \kappa_1$ and $\kappa \rightarrow \kappa_2$ then there exists κ_1' and κ_2' such that $\kappa_1 \rightarrow^? \kappa_1'$ and $\kappa_2 \rightarrow^? \kappa_2'$ where $\kappa_1' \approx \kappa_2'$.*

We write $\rightarrow^?$ to denote the reflexive closure of $\rightarrow$—i.e., $\kappa \rightarrow^? \kappa'$ iff $\kappa \rightarrow \kappa'$ or $\kappa = \kappa'$. Equivalence between configurations ($\kappa_1 \approx \kappa_2$) is taken to be up to graph equivalence and consistent renaming of variables and node names in both the graph and the constraint set.

Proof sketch. The proof is by case analysis on all critical pairs of possible-reductions $\kappa \rightarrow \kappa_1$ and $\kappa \rightarrow \kappa_2$. As usual, many of them are trivial. Others require some insight, such as critical pair of two different equality constraints, where the diamond is constructed by making use of the fact that the computed unifiers are most general.

A worthwhile critical pair to consider is the pair of a query simplification (OP-QUERY-GUARDED) in one branch and an edge-assertion simplification (OP-EDGE) in the other. This case and the construction of the diamond is summarized in figure 6.19.

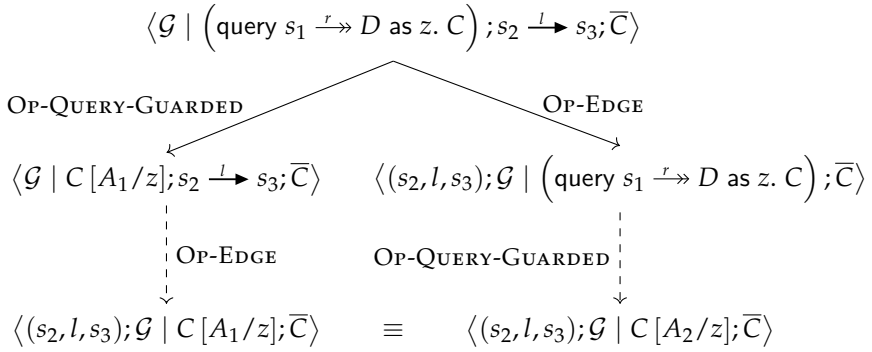


Figure 6.19: Diamond for the critical pair of a query reduction and an edge assertion.

The reduction steps that we have as premises are given by the solid arrows while the dashed arrows represent reduction steps to the existentially quantified configurations of the lemma: Where we use:

$$A_1 = \mathsf{Ans}\left(\mathcal{G}, s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right) \qquad A_2 = \mathsf{Ans}\left((s_2, l, s_3); \mathcal{G}, s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right)$$

The diamond is formed by applying the same steps in the opposite branch, as usual. The resulting states are identical if we can prove the following equality:

$$\mathsf{Ans}\left(\langle S, E, \rho\rangle, s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right) = \mathsf{Ans}\left(\langle S, (s_1, l, s_3); E, \rho\rangle, s_1 \xrightarrow{r}\!\!\!\!\rightarrow D\right)$$

Here (s_2, l, s_3) is the newly asserted edge. The guard on query simplification guarantees that the new edge is not weakly critical for the query. Hence by lemma 4 we get that the equality holds. ▲ □

Confluence of arbitrary reduction sequences is then provable using the diamond property in the usual way:

Theorem 13 (Confluence)**.** *If $\kappa \to^* \kappa_1$ and $\kappa \to^* \kappa_2$ then there exists κ_1' and κ_2' such that $\kappa_1 \to^* \kappa_1'$ and $\kappa_2 \to^* \kappa_2'$ where $\kappa_1' \approx \kappa_2'$.*

Proof. The property follows by rule induction on $\to^*$ and a standard "strip lemma" which says that for any $\kappa, \kappa_1, \kappa_2$, if $\kappa \to^* \kappa_1$ and $\kappa \to \kappa_2$ then there exists a κ_1' and κ_2' such that $\kappa_1 \to^* \kappa_1'$ and $\kappa_2 \to^* \kappa_2'$ where $\kappa_1' \approx \kappa_2'$. □

6.5 Implementation and Case Studies

We developed the operational semantics of Statix-core and have proven that the operational semantics computes sound name resolution results for well-formed specifications. However, the well-formedness restriction and the possibility that the scheduling gets

stuck limits the expressiveness of Statix-core. In this section we describe an evaluation of our approach using MiniStatix: a prototype implementation of Statix that closely follows the operational semantics.

MiniStatix implements the core constraint language Statix-core, as well as (mutually) recursive predicates and (guarded) pattern matching, in approximately 3000 lines of Haskell. The language has a simple module system to enable the larger case study language specifications to be organized across files. After parsing, the specification is statically checked: names are statically resolved, after which *permissions are inferred* for constraints, deriving the relations formally stated in figure 6.17 and figure 6.18. The implementation extends the definition of permissions and well-formedness to predicates and pattern matching.

The solver implementation is a variation of the small-step operational semantics that uses environments rather than substitution. It uses a round-robin, delaying scheduler for constraints, which can detect configurations where no more progress can be made (i.e., stuckness). For satisfied constraints, the solver outputs a complete scope graph and the unifier for the top-level existential quantifier if there is any. For rejected programs, the solver will give the trace of instantiated predicates that led to falsification, which functions as a formal explanation of the error. Stuck configurations are output for specification debugging purposes.

We have evaluated our approach using MiniStatix on three case studies by implementing a subset of name resolution for Java and Scala, and the whole of LMR[24,25]. The former two show that our approach can indeed resolve challenging patterns of real languages. By targeting subsets of real languages, we are able to directly test our approach against the Java and Scala type checker. The test succeeds if MiniStatix and the reference type checker agree on whether a test program is valid. Programs that should be rejected are equipped with specific error expectations to avoid false positives. The third case study (LMR) is used to explain when our approach is incomplete, causing stuck configurations in MiniStatix. We count a test case as a success if it does not get stuck and meets the manually set test expectation (because LMR has no reference type checker). The results are summarized in table 6.20 and we briefly highlight some parts of the case studies below.

[24] Van Antwerpen et al. 2016. "A constraint language for static semantic analysis based on scope graphs"

[25] A toy *Language with Modules and Records*.

Language	LOC Spec	Tests	Succeed	Fail	Stuck
Java	1201	125	125	0	0
Scala	517	109	109	0	0
LMR	263	19	15	0	4
Total	1976	253	249	0	4

▲

Figure 6.20: Test numbers and results from our case studies.

For the Java study we selected a subset of Java with a focus on the binding aspects of packages, imports, classes, interfaces, inheritance, inner classes, and method and field members. Test cases are set up so that faulty name resolutions result in type errors and focus on interesting edge cases. The tests come in pairs that test that good programs are accepted and ill-typed variants are rejected.

Packages in Java are an interesting test subject because at first sight they seem to require *remote extension*—i.e. the very pattern that is forbidden by our well-formedness restriction. Package names in Java have no authoritative declaration, but exist by virtue of use. More than one compilation unit can define members in the same package. The well-formedness restriction indeed does not permit modeling this by resolving the package name at the top of a compilation unit to obtain a package scope and contributing definitions to that scope. This would constitute *remote extension* of the package scope. However, the right binding semantics can also be modeled via a mixin-pattern: compilation units query for all other compilation units in the same package and make their types accessible by adding import edges. This model makes it locally very apparent what things are in scope of the compilation unit, and also passes the well-formedness check so that stability of query answers can be guaranteed.

The Scala study focuses on the resolution of local definitions and imported names. Scala not only gives different precedence levels to local definitions, wildcard, and specific imports, but also distinguishes their scope. Concretely, local definitions are accessible in the *surrounding* scope to accommodate mutual definitions, whereas imported names are only accessible in *subsequent* scope. This ensures that resolving import statements cannot influence their own resolution. This simplifies scheduling because it avoids the need to iterate name resolution within a block. We discuss iterated name

resolution (which Rust and LMR require) in more detail below.

The following well-typed example test case highlights the scoping difference between declarations and imports, and also shows specific imports, wildcard imports, and imports from imported objects.

```
object c {
  import a._;
  def g(): Unit = {
    val x: Int = h();
    import b.h;
  };
  def h(): Int = 42;
};
object a {
  object b {
    def h(): Unit = {};
}; };
```

◀

Figure 6.21: Example test case from the Scala case study. Our simple parser only accepts semicolon-terminated statements.

The forward reference to the locally defined object `a` is well bound, whereas the imported definition of `h` cannot be forward referenced. In addition to the features shown, our Scala subset supports hiding and renaming in imports.

In the LMR/Rust study we looked at imports that can affect their own resolution. Although this does not appear to be a common language feature, Rust, at least, does implement this semantics. The difficulty arises because LMR and Rust combine features that are not usually found together in other module systems: relative imports, unordered imports, and glob imports. The combination of these features make programs such as the following well-typed.

```
pub mod foo {
  pub mod bar {}
}

pub mod test {
  use super::*;
  use bar::*;
  use foo::*;
}
```

◀

Figure 6.22: Well-typed Rust example.

While Scala imports also resolve relative to their local scope, they only open in *subsequent* scope—i.e., they are *ordered*. The direct Scala

equivalent of the given example would not resolve the name `bar`. The following example program shows how self-influencing import can become confusingly ambiguous.

```
pub mod foo {
  pub mod foo {}
}

pub mod test {
  use super::*;
  use foo::*;
}
```

◀

Figure 6.23: Ambiguous Rust example.

The Rust type checker judges this program to be ambiguous: because imports do not shadow outer declarations, two declarations of `foo` are visible in the block of module `test`.

The Rust type checker uses iterated name resolution to implement the desired behavior, re-resolving module names until the environment stabilizes. MiniStatix on the other hand gets stuck on Rust/LMR programs with imports—i.e., also non-ambiguous programs. The import is specified using a query and an import edge assertion. However, the query is delayed on the weakly critical edge assertion that in turn is waiting on the query to resolve the target scope of the edge.

The difference between Scala's and Rust's imports exactly exposes the limits of our particular over-approximation of dependencies using *weakly* critical edges: it may lead to the operational semantics being stuck on programs that in principle have a stable model. Rust shows that a sound fixed point algorithm exists for name resolution in Rust programs. How to systematically derive such an algorithm from high-level declarative specifications is a different question. From a declarative specification of self-influencing imports some paradoxes can arise. It is worth pondering what should be the meaning of the program in figure 6.23 if imports *were* to shadow outer declarations.

6.6 *Related Work*

The main novelties of the Statix specification language compared to typical typing rules are the assertions of scope graph structure, and the queries over the resulting graph. The fact that scopes are passed

by reference enables the high-level specification of name binding in two ways. First, it makes it possible to separate the assertion that a scope exists from the description of its contents. This is useful because scope is naturally a concept that extends over larger parts of syntax, whereas typing rules are usually given by induction over the syntax. Second, it makes retrieving binding information about remote parts of the AST lightweight, because this information is accessible via scope references. This makes it unnecessary to propagate and construct complicated environments in typing rules.

At the same time, these features present a challenge operationally. In order to maintain soundness with respect to the declarative semantics, queries need to be delayed until all contributions to the relevant scopes have been witnessed. This chapter addresses that challenge. In this section, we want to relate to and compare with other approaches to operationalizing declarative specifications of static semantics.

Constraint Generation and Solving. Statix is a constraint language in the tradition of Constraint Handling Rules (CHR)[26]. CHR has a sound semantics of fact assertion and retraction. Fact assertion and retraction are considered impure primitives in Prolog[27]. Where CHR uses the constraint store to record assertions, Statix uses the scope graph. Unlike constraint store facts, scope graph facts are only asserted and never retracted. The context-sensitive effects that can be achieved using multi-head simplification and propagation rules in CHR can be realized using scope graph constraints in Statix.

The approach of CHR and Statix is distinctly different from approaches that separate the constraint generation and constraint solving phases in the tradition of Hindley-Milner type-inference[28,29]. The constraint-generation based formalism that is closest to Statix is its precursor NaBL2[30]. Like Statix, it has built-in support for name resolution using scope graphs[31], but separates constraint generation from constraint solving.

NaBL2 supports type-dependent name resolution, in which the resolution of a name (such as the method name in $e.m()$) depends on the resolution of a type (for the receiver expression e), which in turn may depend on name resolution. It has to deal with the fact that sometimes not all binding information is available when a name is resolved. The incomplete information is represented explicitly in the model using an *incomplete scope graph*, where unification variables

[26] Frühwirth 1998. "Theory and Practice of Constraint Handling Rules"

[27] Moss 1986. "Cut and Paste—defining the impure Primitives of Prolog"

[28] Odersky et al. 1999. "Type Inference with Constrained Types"

[29] Pottier et al. 2005. "The Essence of ML Type Inference"

[30] Van Antwerpen et al. 2016. "A constraint language for static semantic analysis based on scope graphs"

[31] Neron et al. 2015. "A Theory of Name Resolution"

can be placeholders for scopes. During constraint solving, such unification variables must be unified before they can be traversed as part of queries. The solver guarantees query stability by relying on a resolution algorithm that *delays* when resolution encounters an edge to a unification variable.

Unlike in Statix, scope graphs in NaBL2 can *only* be incomplete in the sense that the target of an edge is yet unknown. Edges cannot be missing entirely. This prohibits specifications where the presence of edges is dependent on resolution in the scope graph. In Statix this is permitted and used[32]. For example, imports-with-hiding in our Scala case study is specified using a query that finds all members of an object scope and a new scope that is a masked version of the object scope. The number of edges of the masked scope depends on query resolution.

[32] Van Antwerpen et al. 2018. "Scopes as types"

On-demand Evaluation of Canonical Attribute Grammars. Another way to operationalize a type system is to use an *attribute grammar* (AG), using equations on AST nodes to define the values of *attributes*. Attributes are either inherited (i.e., computed by the parent and propagated down the AST), or synthesized (i.e., computed on the node itself and propagated upwards). Name resolution can be specified using AGs by taking environment-based typing rules such as in figure 6.4 and turning the downwards and upwards propagating environments into inherited and synthesized attributes respectively.

Canonical attribute grammars were implemented by statically computing a schedule (or plan) consisting of multiple passes over the AST, ordered such that the input values of the attribute computations in one pass are computed in a previous pass (see Alblas[33] for a survey). Expressivity of canonical attribute grammars is limited by this stratified evaluation. By building on the circular programming techniques of Bird[34], Johnsson[35] shows how dependencies between attributes can be determined dynamically, relaxing the non-circularity requirements on specifications. Modern attribute grammar formalisms like JastAdd[36,37] and Silver[38] use these techniques, relying mostly on on-demand computation.

The *specification* problems that we describe in section 6.1 with environment-based rules also affect canonical attribute grammars. In particular, to gain access to binding information from somewhere else in the tree, this information needs to be aggregated and distributed through the least common ancestor[39]. This leads to more

[33] Alblas 1991. "Attribute Evaluation Methods"

[34] Bird 1984. "Using Circular Programs to Eliminate Multiple Traversals of Data"

[35] Johnsson 1987. "Attribute grammars as a functional programming paradigm"

[36] Ekman et al. 2005. "Modular Name Analysis for Java Using JastAdd"

[37] Ekman et al. 2007a. "The JastAdd extensible Java compiler"

[38] Wyk et al. 2010. "Silver: An extensible attribute grammar system"

[39] Boyland 2005. "Remote attribute grammars"

complex, non-modular grammars for languages with complex binding rules[40]. This specification problem is the motivation for *Reference Attribute Grammars* (RAGs), which we discuss separately below.

Boyland[41] also describes how canonical AGs suffer from an *implementation* problem: packaging multiple values into environment attributes requires that they can be computed at the same time. Sometimes this causes circular dependencies that disappear when values are split across multiple environments. This means that the specification writer has to be aware of the operational semantics. Boyland 2005 concludes: "The decision of whether two values can be packaged together (thus reducing complexity and increasing efficiency) relies on global scheduling information, and thus should be left to an implementation tool, not the description writer." This motivates the development of *Remote Attribute Grammars*. The same problem also motivated the design of Statix.

[40] Hedin 2000. "Reference Attributed Grammars"

[41] Boyland 2005. "Remote attribute grammars"

Scheduling of Reference Attributed Grammars with Collection Attributes. Reference attributes[42] are an extension of canonical AGs that allow attributes that *reference* AST nodes. Attributes of the referenced AST nodes can be read directly. This can be used to avoid the need to propagate information using environments, and thus avoids some of the problems with the specification and the implementation of static semantics using environments that we described in section 6.1. Reference attributes can be used to superimpose graphs on an AST.

By themselves, reference attributes do not solve the problems with the aggregation of binding described in section 6.1. To additionally avoid the specification overhead of aggregating values from an AST, they can be combined with parameterized attributes or collection attributes[43].

Parameterized attributes are used for example to define name resolution for large subsets of Java in the JastAdd AG system[44]. This is accomplished by defining a parameterized `lookup` attributes on nodes that implement the name resolution policy. These attributes are invoked on references, passing the name to be resolved. Shadowing can be implemented by deferring to the lookup of child and parent nodes in a particular order. The effective resolution policy for the resolution of a variable is thus determined by the combination of all local policies implemented in the nodes that are traversed. This differs significantly from Statix specifications, where the resolution policy is determined more uniformly by the query parameters in the

[42] Hedin 2000. "Reference Attributed Grammars"

[43] Boyland 1996. *Descriptional composition of compiler components*

[44] Ekman et al. 2007b; Ekman et al. 2007a. "The JastAdd system—modular extensible compiler construction"; "The JastAdd extensible Java compiler"

variable rule. The separation queries from scope graph construction in Statix is designed to make it easy to extract an abstract model of binding. Parameterized attributes are evaluated on-demand.

Collection attributes collect *contributions* that can come from different contributor nodes throughout the AST. A contributor uses a reference attribute to specify to which collection it contributes. The mutual binding example in figure 6.3 can be specified using reference and collection attributes. A block defines a collection attribute that collects the binding contributions from its immediate children. To that end the children need a reference to the block, which can be specified as an inherited attribute. We are not aware of any case studies involving non-lexical static binding that make use of collection attributes for name resolution.

There are two approaches to evaluating AGs with collection attributes. The first approach is due to E. Magnusson et al.[45]. Before a collection is read, all contributions must have been computed. To be able to determine if this is the case, a pass is made over the AST and for all contributions to *any* instance of the collection attribute, the reference that is contributed to it is evaluated. Like in Statix, this is an over-approximation of dependencies. After this, all contributions are evaluated for the one reference whose collection is being read. Because of the first pass, the reference attribute can never depend on any instance of the collection attribute, or a cycle would occur (E. Magnusson et al. 2009). This can cause evaluation to get stuck even when sound schedules exist.

[45] E. Magnusson et al. 2009. "Demand-driven evaluation of collection attributes"

The specification of contributions differs from the specification of edges in Statix, in that edge assertions can occur anywhere in a specification on any scope reference. In a Statix specification that does not enforce our permission-to-extend restriction, it is *not* possible to demand the evaluation of the scope reference that the edge is 'contributed to'. This is the case because the scope reference can be determined by arbitrary constraints, which can be blocked. On the other hand, if permission-to-extend *is* enforced, then it is unnecessary to evaluate all scope references that are contributed to. This is the case because a scope that is not yet ground cannot be instantiated to any existing scope—hence $(x \xrightarrow{l} t) \not\mapsto (s, l)$ is sound.

A Statix specification has no immediate counterpart as a RAG. An obstacle is that Statix rules do not clearly distinguish inputs and outputs, which is part of their declarative appeal. It also potentially

enables them to be used to solve other language implementation problems that involve the static semantics, such as suggesting well-typed program completions[46]. Attribute grammars on the other hand organize specifications into equations for attributes, which have a clear direction. A benefit of this approach is that dependencies are more explicitly present in the specification (even for equations that specify contributions to collection attributes), so that on-demand evaluation is available. Encoding Statix rules into AG equations requires a factorization into attributes. Whether this is always possible is an interesting open research problem.

[46] Pelsmaeker et al. 2019. "Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)"

6.7 Conclusion

We envision closing the gap between language specification and language implementation by using meta-languages that can address the complexity of actual programming languages and systematically deriving implementations from specifications. Importantly, this moves the question of implementation correctness from the concrete language to the meta-language. This approach leads to correct-by-construction language implementations and higher-level specifications that abstract from operational concerns.

In this chapter, we tackled one aspect of that challenge. Critical edges represent language independent insight into a scheduling problem that type checker implementations need to address. Because it is a high-level concept, it can be used to think about language design. We exploit this insight and obtain sound-by-construction scheduling in type checkers derived from specifications.

CONCLUSIONS

7 Conclusions

In this thesis we described the vision that language specifications and/or implementations ought to be written in meta-languages that ensure the relevant properties by construction. This vision shifts our perspective from meta-theory as a *subject* of research, to meta-theory as the *means* to produce a correct implementation of a programming language. As we shift our perspective, requirements change, bringing many new and interesting challenges to the table for meta-languages.

We focused on the properties of type checkers, interpreters, and compilers that ensure that the static semantics of a given language is a meaningful contract between the programmer and the machine. The first thing to be done is to define this contract precisely and declaratively—i.e., abstracting as much as possible over operational concerns. Many language features that are prevalent in deployed programming languages, however, are difficult to specify declaratively using existing formal methods, even though they might be sufficient for idealized object languages.

This is no surprise: many of the features of "surface" languages—and especially also their co-existence, even when orthogonal—are largely ignored in idealized or "core" languages. Hence, we cannot expect that researching meta-theory of individual language features is enough to accommodate formal specification and verification of deployed, general purpose programming languages. Going from "well-understood" to easy to specify and implement correctly by a language developer requires dedicated attention from meta-language designers.

With that in mind, this thesis studied three different commonplace language features (monotone state, linear references, and non-lexical static binding) of programming languages and proposes new

methods and means to concisely specify them. We showed that specifications of static semantics can be greatly improved if we pick an appropriate meta-language that abstracts over the bookkeeping associated with various forms of binding. Not only did we apply this to surface languages, but also to the implementations of their back-ends, typing interpreters and compilers.

Our approach to these implementations is a mix of syntactic techniques and functional programming idioms, exploiting the specification power of a dependently typed host language like Agda. Despite taking a syntactic perspective on type safety, our inspiration for suitable abstractions often stems from more semantic approaches.[1,2] Semantic proofs typically rely on logical abstractions to state and prove type safety because it is unwieldy to directly work with the semantic model.[3] For this reason, the focus on constructing more powerful logics that act as a meta-language for their specifications is much more prevalent there than in the work on syntactic safety. Our results suggest that there is more to be gained by blurring the line between the two approaches.

We found that a dependently typed language is a suitable petri dish for developing shallowly embedded logical meta-languages. The specification power of dependent types is well suited for integrating type invariants into language back ends. At the same time, we are only beginning to understand how to leverage this power. We must satisfy at the same time our hunger for high-level specifications and implementations, but also the demand of the host-language's type checker for definitions that are evidently correct. To find definitions that live in the intersection of these requirements requires much experimentation. This thesis contributes both new ideas for delivering on these requirements and also evidence that existing ideas can scale to programming languages with more complicated type invariants.

The identification of co-contextual typing relations as a generalization of McBride's co-de-Bruijn representation of lexical binding[4] is a very promising new idea in that regard. Co-contextual typings deliver inherently much more localized representations of invariants than conventional contextual typings. These localized invariants fit well with dependent pattern matching in a dependently typed host language such as Agda: inspecting data reveals locally relevant static information. This benefits both the language developer—who is guided towards a correct implementation by these invariants—

[1] Appel et al. 2007. "A very modal model of a modern, major, general type system"

[2] Jung et al. 2018a. "RustBelt: Securing the foundations of the Rust programming language"

[3] Good examples are *step-indexed* models (Dreyer et al. 2011), where working directly with step-indexed propositions requires tedious reasoning about step-indexes

[4] McBride 2018. "Everybody's got to be somewhere"

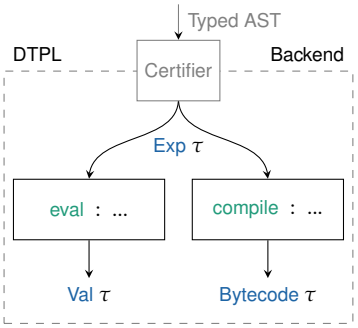
and also the type checker of the host language Agda—which can automatically verify these local invariants more easily than global ones. At the same time, we have seen that the bookkeeping of co-contextual typings can be largely hidden from view using an embedded separation logic. This delivers high-level, declarative specifications with elegant rules, achieving one of the main goals of this thesis.

USING SEPARATION LOGIC as a meta-language for specifying static semantics is not solely applicable to language back-ends in a dependently typed host language, however. In part II of this thesis, we have seen that similar ideas can be used to give meaning to a new specification language for static semantics of languages with non-lexical static binding—delivering Statix-core. We affirmatively answered the question whether such high-level, declarative specifications can automatically be operationalized as type-checkers and proven sound once-and-for-all.

Such a fully automated approach to obtaining a language implementation from a specification is a powerful prototyping tool. The correct-by-construction prototype can be used as a low-cost reference implementation. This is more than mere convenience. If we want programming languages with formal specifications, then we need to give language developers a reason to invest in learning the trade. A low-effort prototype is a direct return on this investment that encourages verified correct implementations before unverified manual implementations set a standard that cannot be recovered from.

7.1 *Future Work*

Where do we go from here?



Bridging the gap between part I and part II. In this thesis we presented different approaches for the implementation of safe front-ends and back-ends of programming languages. We used a different approach for type checkers, because we were confident that we could automatically derive type checkers from specifications, rather than having to manually implement them in a language like Agda. As a result, however, there is now a gap between the specification and implementation of the front-end and that of the back-end. This gap

consists of the need for two specification of static semantics: a Statix specification and an intrinsically typed representation Exp. To connect a Statix front-end to an intrinsically typed back-end, one would need to implement a certifier that converts from the annotated AST produced by Statix to a term that inhabits the typed family Exp.

It would be satisfying if we could bridge the gap between the two parts of this thesis and eliminate the need for two separate specifications. Not just because this would eliminate the overhead, but also because we want to implement type-safe language back-ends for language with non-lexical static binding.

A first attempt to achieve the second goal was described in the conference paper edition[5] of chapter 3, where we gave an account of the global static binding of Middleweight Java (MJ) in Agda. Middleweight Java supports classes with inheritance and access of inherited members. Our approach was to give a *contextual* account of the static binding using scope graphs as a model for structured symbol tables. Although this proved sufficient for type-safe interpretation of MJ, the resulting typed syntax cannot easily be transformed in ways that alter the static binding in the program. This is the case for essentially the same reason as why global label contexts proved difficult in chapter 5: it is difficult to relate a global context to a local view on part of a program. A *co-contextual* specification of global binding may again help solve these problems. One path towards that goal could be to give a co-contextual interpretation of a Statix-like constraint language. Once we have a better idea of how we should represent intrinsically typed terms with non-lexical static binding we can certainly consider automatic delivery of such terms by the Statix solver.

[5] Poulsen et al. 2018. "Intrinsically-typed definitional interpreters for imperative languages"

Correct-by-construction effect-sensitive languages. Many general purpose programming languages (e.g., Java and Rust), but also low-level languages (like JVM bytecode), have type invariants that rely on data-flow. Such data-flow sensitive analyses are sometimes called *effect systems*, distinguishing them from type systems. For example, both the Java type checker and the JVM bytecode verifier statically determine initialization of local variables and reject code that accesses local variables before initializing them. Such properties are relevant for type-safety and type-correct compilation. A back-end that does not observe variable initialization can fail to initialize variables even if the source code does this correctly. To address this in

intrinsically typed language back-ends we need term representations that capture those data-flow-sensitive properties.

Traditionally, data-flow analyses are often specified as fixed-point analyses over graph-based representations of programs,[6] rather than inductively on the AST. Such graph-based specifications would be difficult to integrate into an AST, and local reasoning about a global solution to a fixed point analysis is also complicated. An initial investigation of the problem hints that it may be possible to instead give inductive specifications of some effect analyses in a co-contextual style. This may enable intrinsically effect-annotated expressions that can be manipulated without the need to reason about the non-local impact on data-flow properties. It is an open question whether we can write elegant functions over such expressions in order to implement, for example, intrinsically typed compilers that are guaranteed to produce bytecode that correctly initializes local variables. The specification technique may also have individual merit.

[6] Nielson et al. 1999. "Principles of program analysis"

Intrinsically Typed Interpretation of Garbage Collected References. In chapter 3 we presented an approach using monotone predicates for implementing an intrinsically typed interpreter for a language with references. Hereby we followed the traditional syntactic type-safety proof for such a language, which relies on the observation that programs monotonically extend the store. In practice, however, allocated memory cannot keep growing monotonically without exhausting the finite memory that we own. Hence, such languages are implemented using strategies that rely on *garbage collection* to collect cells that are no longer referenced by the running program.

Using the approach that we presented, we cannot verify an implementation that performs garbage collection, both because this breaks monotonicity, and because there is no real way to determine whether cells are still referenced. Using the insight of chapter 5, we now recognize the traditional typing of monotone state in chapter 3 as being given in a contextual style. Interestingly, it is also possible to give a co-contextual account of type-safe references by instantiating the state monad from chapter 4 with a non-linear store-type proof-relevant separation algebra. Remarkably, this enables the state monad to keep track of whether there are clients of a store cell. This also enables the monad to collect the cell if no clients are left, because monotonicity is not required. Whether this is a useful specification and implementation of garbage-collected references is

up for investigation.

Performance. Throughout this thesis we focused on developing language implementations that satisfy their safety criteria. A whole other dimension of what is yet to be investigated are non-functional properties of our implementations. For example, the performance of Statix type-checkers, as well as the performance of intrinsically typed interpreters and compilers. This is relevant especially if we want to use these implementations for more than just prototyping and reference implementations.

For Statix type checkers, there is evidence that it is possible to automatically derive concurrent type checkers. There is also hope that the type checkers can be made incremental.[7] These avenues are being investigated by my co-authors.

[7] Aerts 2019. *Incrementalizing Statix: A Modular and Incremental Approach for Type Checking and Name Binding using Scope Graphs*

For intrinsically typed programs there are two interesting open-ended research problems related to performance. Firstly, dependently typed languages that mix programs and proofs typically rely on the compiler to *erase* computationally irrelevant values and proofs in order to gain runtime performance. In this thesis, however, we purposefully exploited the integration of programs and proofs to the extent that perhaps all proofs have become computationally relevant. It is unclear to what extent erasure is still applicable to our programs and more generally what the performance characteristics are of our intrinsically typed implementations.

Secondly, we have used rather simplistic data-structures for the implementation of, for example, stores and bytecode sequences. In particular, we primarily used simple list-like data types to represent most collections, and focused our efforts entirely on the representation of invariants of this data. For more realistic interpreters and compilers we likely need more performance-oriented data structures. It may also be necessary to adapt our implementations to use more performant algorithms. It is an interesting research question whether our techniques and tools for representing and abstracting over invariants is compatible with these demands.

Accessibility. Another subject that deserves individual attention is the accessibility of the ideas and methods delivered in this thesis. Much of our progress towards the described vision consists of new ideas on how to use dependently typed languages and is embodied

by Agda libraries with new abstractions. Although we demonstrated that these artifacts *can* improve specifications and implementations of languages, we do not expect that our methods can readily be adopted by the audience of language developers that we aim to support. To achieve our vision we must actively look to make our formal methods approachable for this audience.

The approach of part II with Statix does a better job in that regard. In contrast to the shallowly embedded logics in Agda, a domain specific specification language can hide its implementation details well. The errors reported by a domain specific language like Statix are more likely to be high-level and meaningful to developers. Statix is also integrated with the grammar specification formalism SDF3[8] and the term transformation language Stratego[9,10], and is supported by a specialized development environment[11]. This has already enabled its use in the classroom and in research and development projects in industry.

A major challenge for extracting domain specific specification languages from our Agda libraries is to determine the scope of the language. The boundary of shallowly embedded abstractions is soft, enabling a gradual transition from high-level interfaces to their low-level implementations. We have made good use of this fact. It is not yet clear where one can draw a line that could be the interface of a domain specific language.

[8] Souza Amorim et al. 2020. "Multi-purpose Syntax Definition with SDF3"

[9] Visser et al. 1998. "Building Program Optimizers with Rewriting Strategies"

[10] Bravenboer et al. 2008. "Stratego/XT 0.17. A language and toolset for program transformation"

[11] Kats et al. 2010. "The spoofax language workbench: rules for declarative specification of languages and IDEs"

7.2 *Parting Words*

As I finish the future work section, the keys are starting to come off of my keyboard, reminding me that adding more letters can be subtracting. I hope that the words that do make it into the final cut of this thesis not only explain my technical contributions in the past few years, but also (more than any of the individual papers did) convey the ideas that fueled the research.

Underlying all the work in the thesis is the belief that it is desirable that programming is guided by specification. This is a long-held belief of functional programming, but with the advent of languages like Rust this is now spreading to domains that were previously hard to find suitable type-systems for. Programmers are discovering that type checkers can also be freeing, allowing them to generously use the programming language, without fear of breaking the invariants of the language. Dependently typed languages have the potential

to similarly free programmers from their invariants of their own invention that they want to impose on their data.

This thesis applies this belief to language implementations, using static semantics as the invariant of the programs that we manipulate as data. We saw that many familiar programming devices can be complemented by (perhaps less familiar) logical devices at the type-level. We used this to integrate meta-theory into language implementations using types, closing the gap between specifications and implementations. There are, however, many more ways to bring formal methods out of the realm of things that are perceived to have no use outside one's education, and into the practice of software engineering. Computer programming is one of the very few engineering disciplines where the gap between specification and implementation can be closed. Let's make it happen.

APPENDIX

Bibliography

Abel, A. (2013). "Normalization by Evaluation: Dependent Types and Impredicativity". Habilitationsschrift.

– (2020). "Type-preserving compilation via dependently typed syntax in Agda". Abstract of a talk at TYPES, available online at http://www.cse.chalmers.se/~abela/types20.pdf, slides available online at http://www.cse.chalmers.se/~abela/talkTYPES2020.pdf.

Abel, A. and J. Chapman (2014). "Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types". In: *Workshop on Mathematically Structured Functional Programming (MSFP)*, pp. 51–67. DOI: 10.4204/EPTCS.153.4.

Abel, A. and N. Kraus (2011). "A Lambda Term Representation Inspired by Linear Ordered Logic". In: *International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*. Vol. 71. EPTCS, pp. 1–13. DOI: 10.4204/EPTCS.71.1.

Aerts, T. (2019). *Incrementalizing Statix: A Modular and Incremental Approach for Type Checking and Name Binding using Scope Graphs*. MSc Thesis.

Ahmed, A. J. (2004). "Semantics of types for mutable state". PhD thesis. Princeton University.

Alblas, H. (1991). "Attribute Evaluation Methods". In: *Attribute Grammars, Applications and Systems*. Vol. 545. LNCS. Springer, pp. 48–113. DOI: 10.1007/3-540-54572-7_3.

Allais, G. (2017). "Typing with Leftovers - A mechanization of Intuitionistic Multiplicative-Additive Linear Logic". In: *International Conference on Types for Proofs and Programs (TYPES)*. LIPIcs, 1:1–1:22. DOI: 10.4230/LIPIcs.TYPES.2017.1.

Allais, G., R. Atkey, J. Chapman, C. McBride, and J. McKinna (2018). "A type and scope safe universe of syntaxes with binding: their semantics and proofs". In: *Proceedings of the ACM on Programming Languages* 2.ICFP, 90:1–90:30. DOI: 10.1145/3236785.

Allais, G., J. Chapman, C. McBride, and J. McKinna (2017). "Type-and-scope safe programs and their proofs". In: *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 195–207. DOI: 10.1145/3018610.3018613.

Altenkirch, T., J. Chapman, and T. Uustalu (2015). "Monads need not be endofunctors". In: *Logical Methods in Computer Science* 11.1. DOI: 10.2168/LMCS-11(1:3)2015.

Altenkirch, T. and B. Reus (1999). "Monadic Presentations of Lambda Terms Using Generalized Inductive Types". In: *International Workshop on Computer Science Logic (CSL)*, pp. 453–468. DOI: 10.1007/3-540-48168-0_32.

Amin, N. and T. Rompf (2017). "Type soundness proofs with definitional interpreters". In: *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pp. 666–679. DOI: 10.1145/3093333.3009866.

Ancona, D., F. Damiani, S. Drossopoulou, E. Zucca, et al. (2004). "Even more principal typings for Java-like languages". In: *Workshop on Formal Techniques for Java-like Programs (FTfJP)*.

Van Antwerpen, H., P. Néron, A. P. Tolmach, E. Visser, and G. Wachsmuth (2016). "A constraint language for static semantic analysis based on scope graphs". In: *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*. ACM, pp. 49–60. DOI: 10.1145/2847538.2847543.

Van Antwerpen, H., C. B. Poulsen, A. Rouvoet, and E. Visser (2018). "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), 114:1–114:30. DOI: 10.1145/3276484.

Appel, A. W. (2001). "Foundational proof-carrying code". In: *LICS*, pp. 247–256. DOI: 10.1109/LICS.2001.932501.

– (2006). "Compiling with continuations". Cambridge University Press. ISBN: 978-0-521-03311-4.

Appel, A. W., P.-A. Melliès, C. D. Richards, and J. Vouillon (2007). "A very modal model of a modern, major, general type system". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 109–122. DOI: 10.1145/1190216.1190235.

Atkey, R. (2009). "Parameterised notions of computation". In: *Journal of Functional Programming* 19.3–4, pp. 335–376. DOI: 10.1017/S095679680900728X.

– (2018). "Syntax and semantics of quantitative type theory". In: *LICS*, pp. 56–65. DOI: 10.1145/3209108.3209189.

Augustsson, L. (1998). "Cayenne - A Language with Dependent Types". In: *Advanced Functional Programming (AFP)*. Vol. 1608. LNCS, pp. 240–267. DOI: 10.1007/10704973_6.

Augustsson, L. and M. Carlsson (1999). "An exercise in dependent types: A well-typed interpreter". In: *In Workshop on Dependent Types in Programming, Gothenburg*.

Bélanger, O. S., S. Monnier, and B. Pientka (2015). "Programming type-safe transformations using higher-order abstract syntax". In: *JFR* 8.1, pp. 49–91. DOI: 10.6092/issn.1972-5787/5122.

Benton, N., C.-K. Hur, A. Kennedy, and C. McBride (2012). "Strongly Typed Term Representations in Coq". In: *Journal of Automated Reasoning* 49.2, pp. 141–159. DOI: 10.1007/s10817-011-9219-0.

Bernardy, J.-P., M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack (2018). "Linear Haskell: Practical linearity in a higher-order polymorphic language". In: *Proceedings of the ACM on Programming Languages* 2.POPL, 5:1–5:29. DOI: 10.1145/3158093.

Bird, R. S. (1984). "Using Circular Programs to Eliminate Multiple Traversals of Data". In: *Acta Informatica* 21, pp. 239–250. DOI: 10.1007/BF00264249.

Bird, R. S. and L. G. L. T. Meertens (1998). "Nested Datatypes". In: *International Conference on Mathematics of Program Construction (MPC)*. LNCS, pp. 52–67. DOI: 10.1007/BFb0054285.

Bornat, R., C. Calcagno, P. W. O'Hearn, and M. J. Parkinson (2005). "Permission accounting in separation logic". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 259–270. DOI: 10.1145/1040305.1040327.

Boyland, J. T. (1996). *Descriptional composition of compiler components*. Tech. rep. UCB//CSD-96-916. University of California. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1996/CSD-96-916.pdf.

– (2005). "Remote attribute grammars". In: *Journal of the ACM* 52.4, pp. 627–687. DOI: 10.1145/1082036.1082042.

Brady, E. C. (2013). "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal Functional Programming* 23.5, pp. 552–593. DOI: 10.1017/S095679681300018X.

Brady, E. C. and K. Hammond (2012). "Resource-safe systems programming with embedded domain specific languages". In: *PADL*, pp. 242–257. DOI: 10.1007/978-3-642-27694-1_18.

Bravenboer, M., K. T. Kalleberg, R. Vermaas, and E. Visser (2008). "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1-2, pp. 52–70. DOI: 10.1016/j.scico.2007.11.003.

Brzozowski, J. A. (1964). "Derivatives of Regular Expressions". In: *Journal of the ACM* 11.4, pp. 481–494. DOI: 10.1145/321239.321249.

Calcagno, C., P. W. O'Hearn, and H. Yang (2007). "Local action and abstract separation logic". In: *IEEE Symposium on Logic in Computer Science (LICS)*, pp. 366–378. DOI: 10.1109/LICS.2007.30.

Capretta, V. (2005). "General recursion via coinductive types". In: *Logical Methods in Computer Science* 1.2. DOI: 10.2168/LMCS-1(2:1)2005.

Castro-Perez, D., F. Ferreira, and N. Yoshida (2020). "EMTST: Engineering the Meta-theory of Session Types". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 278–285. DOI: 10.1007/978-3-030-45237-7_17.

Chapman, J. (2009). "Type Theory Should Eat Itself". In: *Electronic Notes in Theoretical Computer Science* 228, pp. 21–36. DOI: 10.1016/j.entcs.2008.12.114.

Chapman, J., R. Kireev, C. Nester, and P. Wadler (2019). "System F in Agda, for Fun and Profit". In: *International Conference on Mathematics of Program Construction (MPC)*. LNCS, pp. 255–297. DOI: 10.1007/978-3-030-33636-3_10.

Chlipala, A. (2007). "A certified type-preserving compiler from lambda calculus to assembly language". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 54–65. DOI: 10.1145/1250734.1250742.

Cockx, J. (2017). "Dependent pattern matching and proof-relevant unification". PhD thesis. Katholieke Universiteit Leuven.

– (2020). "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules". In: *Types for Proofs and Programs (TYPES)*. Vol. 175. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2:1–2:27. DOI: 10.4230/LIPIcs.TYPES.2019.2.

Cockx, J. and A. Abel (2016). "Sprinkles of extensionality for your vanilla type theory". Abstract of a talk at TYPES, available online at https://jesper.sikanda.be/files/sprinkles-of-extensionality.pdf, slides available online at https://jesper.sikanda.be/files/TYPES2016-presentation.pdf.

Coquand, T. (1992). "Pattern matching with dependent types". In: *Informal proceedings of Logical Frameworks*. Vol. 92, pp. 66–79.

Crary, K. (2003). "Toward a foundational typed assembly language". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 198–212. DOI: 10.1145/640128.604149.

Danielsson, N. A. (2012). "Operational semantics using the partiality monad". In: *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 127–138. DOI: 10.1145/2364527.2364546.

De Bruijn, N. G. (1972). "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier, pp. 381–392.

Devriese, D. and F. Piessens (2011). "On the bright side of type classes: instance arguments in Agda". In: *ACM SIGPLAN international conference on Functional Programming (ICFP)*, pp. 143–155. DOI: 10.1145/2034773.2034796.

Dijkstra, E. W. (1972). "Notes on structured programming". In: *Structured Programming*. Academic Press. Chap. I, pp. 1–82.

Dockins, R., A. Hobor, and A. W. Appel (2009). "A fresh look at separation algebras and share accounting". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. Vol. 5904. LNCS, pp. 161–177. DOI: 10.1007/978-3-642-10672-9_13.

Dreyer, D., A. Ahmed, and L. Birkedal (2011). "Logical Step-Indexed Logical Relations". In: *Logical Methods in Computer Science (LMCS)* 7.2. DOI: 10.2168/LMCS-7(2:16)2011.

Dybjer, P. (1994). "Inductive Families". In: *Formal Aspects of Computing* 6.4, pp. 440–465. DOI: 10.1007/BF01211308.

Ekman, T. and G. Hedin (2005). "Modular Name Analysis for Java Using JastAdd". In: *Generative and Transformational Techniques in Software Engineering*. Vol. 4143. LNCS. Springer, pp. 422–436. DOI: 10.1007/11877028_18.

– (2007a). "The JastAdd extensible Java compiler". In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, pp. 1–18. DOI: 10.1145/1297027.1297029.

– (2007b). "The JastAdd system—modular extensible compiler construction". In: *Science of Computer Programming* 69.1-3, pp. 14–26. DOI: 10.1016/j.scico.2007.02.003.

Erdweg, S., O. Bracevac, E. Kuci, M. Krebs, and M. Mezini (2015). "A co-contextual formulation of type rules and its application to incremental type checking". In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 880–897. DOI: 10.1145/2814270.2814277.

Farka, F., A. Nanevski, A. Banerjee, G. A. Delbianco, and I. Fábregas (2021). "On algebraic abstractions for concurrent separation logics". In: *Proceedings of the ACM on Programming Languages* 5.POPL, pp. 1–32. DOI: 10.1145/3434286. URL: https://doi.org/10.1145/3434286.

Felleisen, M. and D. P. Friedman (1987). "Control operators, the SECD-machine, and the λ-calculus". In: *IFIP WG 2.2 Working Conference on Formal Description of Programming Concepts*, pp. 193–222.

Fowler, S., S. Lindley, J. G. Morris, and S. Decova (2019). "Exceptional asynchronous session types: Session types without tiers". In: *Proceedings of the ACM on Programming Languages* 3.POPL, 28:1–28:29.

Frühwirth, T. W. (1998). "Theory and Practice of Constraint Handling Rules". In: *The Journal of Logic Programming* 37.1-3, pp. 95–138. DOI: 10.1016/S0743-1066(98)10005-5.

Gay, S. J. and V. T. Vasconcelos (2010). "Linear type theory for asynchronous session types". In: *Journal of Functional Programming* 20.1, pp. 19–50. DOI: 10.1017/S0956796809990268.

Guillemette, L.-J. and S. Monnier (2008). "A type-preserving compiler in Haskell". In: *ICFP*, pp. 75–86. DOI: 10.1145/1411204.1411218.

Hancock, P. and A. Setzer (2000). "Interactive programs in dependent type theory". In: *CSL*. Vol. 1862. LNCS, pp. 317–331.

Harper, R. (1994). "A Simplified Account of Polymorphic References". In: *Information Processing Letters* 51.4, pp. 201–206. DOI: 10.1016/0020-0190(94)90120-1.

– (2016). "Practical foundations for programming languages". Cambridge University Press.

Harper, R. and C. A. Stone (2000). "A type-theoretic interpretation of standard ML". In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Ed. by G. D. Plotkin, C. Stirling, and M. Tofte. The MIT Press, pp. 341–388. ISBN: 978-0-262-16188-6.

Hedin, G. (2000). "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3.

Hinrichsen, J. K., J. Bengtson, and R. Krebbers (2020). "Actris: Session-type based reasoning in separation logic". In: *Proceedings of the ACM on Programming Languages* 4.POPL, 6:1–6:30. DOI: 10.1145/3371074.

Hinrichsen, J. K., D. Louwrink, R. Krebbers, and J. Bengtson (2021). "Machine-checked semantic session typing". In: *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 178–198. DOI: 10.1145/3437992.3439914.

Hoare, C. A. R. (1969). "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10, pp. 576–580. DOI: 10.1145/363235.363259.

Honda, K., V. T. Vasconcelos, and M. Kubo (1998). "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *European Symposium on Programming (ESOP)*. Ed. by C. Hankin. Vol. 1381. LNCS, pp. 122–138. DOI: 10.1007/BFb0053567.

Jim, T. (1996). "What are principal typings and what are they good for?" In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 42–53. DOI: 10.1145/237721.237728.

Johnsson, T. (1987). "Attribute grammars as a functional programming paradigm". In: *Functional Programming Languages and Computer Architecture*. Vol. 274. LNCS. Springer, pp. 154–173. DOI: 10.1007/3-540-18317-5_10.

Jung, R., J.-H. Jourdan, R. Krebbers, and D. Dreyer (2018a). "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL, 66:1–66:34. DOI: 10.1145/3158154.

Jung, R., R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer (2018b). "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28, e20. DOI: 10.1017/S0956796818000151.

Jung, R., D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer (2015). "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 637–650. DOI: 10.1145/2676726.2676980.

Kats, L. C. L. and E. Visser (2010). "The spoofax language workbench: rules for declarative specification of languages and IDEs". In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 444–463. DOI: 10.1145/1869459.1869497.

Klein, G. and T. Nipkow (2006). "A machine-checked model for a Java-like language, virtual machine, and compiler". In: *ACM Transactions on Programming Languages and Systems* 28.4, pp. 619–695. DOI: 10.1145/1146809.1146811.

Kock, A. (1972). "Strong functors and monoidal monads". In: *Archiv der Mathematik* 23.1, pp. 113–120.

Krebbers, R. (2015). "The C standard formalized in Coq". PhD thesis. Radboud University Nijmegen.

Krebbers, R., A. Timany, and L. Birkedal (2017). "Interactive proofs in higher-order concurrent separation logic". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 205–217. DOI: 10.1145/3093333.3009855.

Kripke, S. A. (1963). "Semantical Considerations on Modal Logic". In: *Acta Philosophica Fennica*.

Krishna, S., D. E. Shasha, and T. Wies (2018). "Go with the flow: compositional abstractions for concurrent data structures". In: *Proceedings of the ACM on Programming Languages* 2.POPL, 37:1–37:31. DOI: 10.1145/3158125.

Kuci, E., S. Erdweg, O. Bracevac, A. Bejleri, and M. Mezini (2017). "A co-contextual type checker for Featherweight Java". In: *European Conference on Object-Oriented Programming (ECOOP)*, 18:1–18:26. DOI: 10.4230/LIPIcs.ECOOP.2017.18.

Kumar, R., M. O. Myreen, M. Norrish, and S. Owens (2014). "CakeML: A verified implementation of ML". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 179–192. DOI: 10.1145/2535838.2535841.

Leroy, X. (2009). "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7, pp. 107–115. DOI: 10.1145/1538788.1538814.

Lindholm, T., F. Yellin, G. Bracha, A. Buckley, and D. Smith (2020). "The Java Virtual Machine specification: Java SE 14 edition". Available online at https://docs.oracle.com/javase/specs/jvms/se14/jvms14.pdf.

Magnusson, E., T. Ekman, and G. Hedin (2009). "Demand-driven evaluation of collection attributes". In: *Automated Software Engineering* 16.2, pp. 291–322. DOI: 10.1007/s10515-009-0046-z.

Magnusson, L. (1994). "The Implementation of ALF—a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution". PhD thesis. Chalmers University of Technology/Göteborg University.

McBride, C. (2004). "Epigram: Practical Programming with Dependent Types". In: *Advanced Functional Programming (AFP)*. Vol. 3622. LNCS, pp. 130–170. DOI: 10.1007/11546382_3.

– (2011). "Kleisli Arrows of Outrageous Fortune".

– (2012). "Agda-curious?" Keynote at ICFP '14, available online at https://dl.acm.org/doi/pdf/10.1145/2364527.2364529.

– (2014). "How to keep your neighbours in order". In: *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 297–309. DOI: 10.1145/2628136.2628163.

– (2018). "Everybody's got to be somewhere". In: *Workshop on Mathematically Structured Functional Programming (MSFP)*. Vol. 275. EPTCS, pp. 53–69. DOI: 10.4204/EPTCS.275.6.

McBride, C. and J. McKinna (2004). "The view from the left". In: *Journal of Functional Programming* 14.1, pp. 69–111. DOI: 10.1017/S0956796803004829.

McKinna, J. and J. Wright (2006). "A type-correct, stack-safe, provably correct, expression compiler". Unpublished draft.

Milner, R. (1978). "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3, pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

Milner, R., J. Parrow, and D. Walker (1992). "A Calculus of Mobile Processes, I". In: *Information and Computation* 100.1, pp. 1–40. DOI: 10.1016/0890-5401(92)90008-4.

Milner, R., M. Tofte, R. Harper, and D. MacQueen (1997). "The Definition of Standard ML, Revised". The MIT Press.

Moggi, E. (1991). "Notions of computation and monads". In: *Information and Computation* 93.1, pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

Morris, J. G. and J. McKinna (2019). "Abstracting extensible data types: or, rows by any other name". In: *Proceedings of the ACM on Programming Languages* 3.POPL, 12:1–12:28. DOI: 10.1145/3290325.

Morrisett, G., K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic (1999a). "TALx86: A realistic typed assembly language". In: *Workshop on Compiler Support for System Software*, pp. 25–35.

Morrisett, G., D. Walker, K. Crary, and N. Glew (1999b). "From system F to typed assembly language". In: *TOPLAS* 21.3, pp. 527–568. DOI: 10.1145/319301.319345.

Moss, C. (1986). "Cut and Paste—defining the impure Primitives of Prolog". In: *International Conference on Logic Programming (ICLP*. Vol. 225. LNCS. Springer, pp. 686–694. DOI: 10.1007/3-540-16492-8_118.

Neron, P., A. P. Tolmach, E. Visser, and G. Wachsmuth (2015). "A Theory of Name Resolution". In: *European Symposium on Programming*. Vol. 9032. LNCS. Springer, pp. 205–231. DOI: 10.1007/978-3-662-46669-8_9.

Nielson, F., H. R. Nielson, and C. Hankin (1999). "Principles of program analysis". Springer. DOI: 10.1007/978-3-662-03811-6.

Nipkow, T. and G. Klein (2014). "IMP: A Simple Imperative Language". In: *Concrete Semantics*. Springer. Chap. 7, pp. 75–94. DOI: 10.1007/978-3-319-10542-0_7.

Norell, U. (2008). "Dependently Typed Programming in Agda". In: *Advanced Functional Programming (AFP)*. Vol. 5832. LNCS, pp. 230–266. DOI: 10.1007/978-3-642-04652-0_5.

O'Hearn, P. W. and D. J. Pym (1999). "The logic of bunched implications". In: *Bulletin of Symbolic Logic* 5.2, pp. 215–244. DOI: 10.2307/421090.

O'Hearn, P. W., J. C. Reynolds, and H. Yang (2001). "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic (CSL)*. Vol. 2142. LNCS. Springer, pp. 1–19. DOI: 10.1007/3-540-44802-0_1.

Odersky, M., M. Sulzmann, and M. Wehr (1999). "Type Inference with Constrained Types". In: *Theory and Practice of Object Systems (TAPOS)* 5.1, pp. 35–55. DOI: 10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4.

Orchard, D., V.-B. Liepelt, and H. Eades III (2019). "Quantitative program reasoning with graded modal types". In: *Proceedings of the ACM on Programming Languages* 3.ICFP, 110:1–110:30. DOI: 10.1145/3341714.

Pardo, A., E. Gunther, M. Pagano, and M. Viera (2018). "An internalist approach to correct-by-construction compilers". In: *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, 17:1–17:12. DOI: 10.1145/3236950.3236965.

Paterson, R. (2001). "A new notation for arrows". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP*, pp. 229–240.

Paykin, J. and S. Zdancewic (2017). "The linearity Monad". In: *ACM SIGPLAN International Symposium on Haskell*, pp. 117–132. DOI: 10.1145/3122955.3122965.

Pelsmaeker, D. A. A., H. van Antwerpen, and E. Visser (2019). "Towards Language-Parametric Semantic Editor Services Based on Declarative Type System Specifications (Brave New Idea Paper)". In: *European*

Conference on Object-Oriented Programming (ECOOP). Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. DOI: 10.4230/LIPIcs.ECOOP.2019.26.

Pierce, B. C. (2002). "Types and programming languages". MIT Press.

Plotkin, G. D. and J. Power (2002). "Notions of Computation Determine Monads". In: *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, pp. 342–356. DOI: 10.1007/3-540-45931-6_24.

Pollack, R. (1995). "The theory of LEGO". PhD thesis. University of Edinburgh, UK. URL: http://hdl.handle.net/1842/504.

Pottier, F. and D. Rémy (2005). "The Essence of ML Type Inference". In: *Advanced Topics in Types and Programming Languages*. The MIT Press, pp. 389–489.

Poulsen, C. B., P. Néron, A. P. Tolmach, and E. Visser (2016). "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics". In: *European Conference on Object-Oriented Programming (ECOOP)*, 20:1–20:26. DOI: 10.4230/LIPIcs.ECOOP.2016.20.

Poulsen, C. B., A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser (2018). "Intrinsically-typed definitional interpreters for imperative languages". In: *Proceedings of the ACM on Programming Languages* 2.POPL, 16:1–16:34. DOI: 10.1145/3158104.

Reynolds, J. C. (2000). "The meaning of types from intrinsic to extrinsic semantics". In: *BRICS Report Series* 7.32.

– (1972). "Definitional interpreters for higher-order programming languages". In: *ACM annual conference*. Republished in 1998., pp. 717–740. DOI: 10.1145/800194.805852.

– (1998a). "Definitional Interpreters for Higher-Order Programming Languages". In: *Higher-Order and Symbolic Computation* 11.4, pp. 363–397. DOI: 10.1023/A:1010027404223.

– (1998b). "Theories of programming languages". Cambridge University Press. ISBN: 978-0-521-59414-1.

– (2002). "Separation logic: A logic for shared mutable data structures". In: *IEEE Symposium on Logic in Computer Science (LICS)*, pp. 55–74.

Rouvoet, A., H. van Antwerpen, C. B. Poulsen, R. Krebbers, and E. Visser (2020a). "Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications". In: *Proceedings of the ACM on Programming Languages* 4.Object-Oriented Programming Systems, Languages & Applications (OOPSLA), 180:1–180:28. DOI: 10.1145/3428248.

Rouvoet, A., C. Bach Poulsen, R. Krebbers, and E. Visser (2020b). "Intrinsically-typed definitional interpreters for linear, session-typed languages". In: *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 284–298. DOI: 10.1145/3372885.3373818.

Rouvoet, A., R. Krebbers, and E. Visser (2021). "Intrinsically-typed compilation with nameless labels". In: *Proceedings of the ACM on Programming Languages* 5.POPL, pp. 1–28. DOI: 10.1145/3434303.

Rust developers (2020a). "About declarative specifications of Rust's name resolution". https://github.com/nrc/rfcs/blob/name-resolution/text/0000-name-resolution.md.

– (2020b). "About Rust's name resolution". https://github.com/nrc/rfcs/blob/name-resolution/text/0000-name-resolution.md.

Siek, J. G. (May 2013). "Type Safety in Three Easy Lemmas". http://siek.blogspot.co.uk/2013/05/type-safety-in-three-easy-lemmas.html.

Souza Amorim, L. E. de and E. Visser (2020). "Multi-purpose Syntax Definition with SDF3". In: *International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 1–23. DOI: 10.1007/978-3-030-58768-0_1.

Stump, A. (2016). "Verified functional programming in Agda". ACM.

Swamy, N., J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits (2013). "Verifying higher-order programs with the Dijkstra monad". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 387–398. DOI: 10.1145/2491956.2491978.

Swierstra, W. (2009a). "A functional specification of effects". PhD thesis. University of Nottingham, UK. URL: http://eprints.nottingham.ac.uk/10779/.

– (2009b). "A Hoare Logic for the State Monad". In: *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pp. 440–451. DOI: 10.1007/978-3-642-03359-9_30.

Swierstra, W. and T. Baanen (2019). "A predicate transformer semantics for effects (functional pearl)". In: *Proceedings of the ACM on Programming Languages* 3.ICFP, 103:1–103:26. DOI: 10.1145/3341707.

Takeuchi, K., K. Honda, and M. Kubo (1994). "An Interaction-based Language and its Typing System". In: *International Conference on Parallel Architectures and Languages Europe (PARLE)*. Vol. 817. LNCS, pp. 398–413. DOI: 10.1007/3-540-58184-7_118.

Thiemann, P. (2019). "Intrinsically-typed mechanized semantics for session types". In: *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, 19:1–19:15.

Timany, A. (2018). "Contributions in Programming Languages Theory: Logical Relations and Type Theory". PhD thesis. KU Leuven.

Timany, A., R. Krebbers, and L. Birkedal (2017). "Logical relations in Iris". In: *International Workshop on Coq for Programming Languages (CoqPL)*.

Vasconcelos, V. T. (2012). "Fundamentals of session types". In: *Information and Computation* 217, pp. 52–70. DOI: 10.1016/j.ic.2012.05.002.

Vasconcelos, V. T., S. J. Gay, and A. Ravara (2006). "Type checking a multithreaded functional language with session types". In: *Theoretical Computer Science* 368.1-2, pp. 64–87. DOI: 10.1016/j.tcs.2006.06.028.

Visser, E., Z.-E.-A. Benaissa, and A. P. Tolmach (1998). "Building Program Optimizers with Rewriting Strategies". In: *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 13–26. DOI: 10.1145/289423.289425.

Wadler, P. (1992). "The Essence of Functional Programming". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 1–14. DOI: 10.1145/143165.143169.

– (1998). "The Marriage of Effects and Monads". In: *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 63–74. DOI: 10.1145/289423.289429.

– (2014). "Propositions as sessions". In: *Journal of Functional Programming* 24.2-3, pp. 384–418. DOI: 10.1017/S095679681400001X.

Wadler, P., W. Kokke, and J. G. Siek (July 2020). "Programming Language Foundations in Agda". URL: http://plfa.inf.ed.ac.uk/20.07/.

Walker, D. (2005). "Substructural type systems". In: *Advanced topics in types and programming languages*. Ed. by B. C. Pierce. Chap. 1, pp. 3–43.

Wells, J. B. (2002). "The essence of principal typings". In: *ICALP*, pp. 913–925. DOI: 10.1007/3-540-45465-9_78.

Wood, J. and R. Atkey (2020). "A Linear Algebra Approach to Linear Metatheory". In: *Computing Research Repository* abs/2005.02247. eprint: 2005.02247. URL: https://arxiv.org/abs/2005.02247.

Wright, A. K. and M. Felleisen (1994). "A Syntactic Approach to Type Soundness". In: *Information and Computation* 115.1, pp. 38–94. DOI: 10.1006/inco.1994.1093.

Wyk, E. V., D. Bodin, J. Gao, and L. Krishnan (2010). "Silver: An extensible attribute grammar system". In: *Science of Computer Programming* 75.1-2, pp. 39–54. DOI: 10.1016/j.scico.2009.07.004.

Yoshida, N. and V. T. Vasconcelos (2007). "Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication". In: *Electronic Notes in Theoretical Computer Science* 171.4, pp. 73–93. DOI: 10.1016/j.entcs.2007.02.056.

Zalakain, U. and O. Dardha (2021). "π with Leftovers: A Mechanisation in Agda". In: *IFIP WG 6.1 Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*. LNCS, pp. 157–174. DOI: 10.1007/978-3-030-78089-0_9.

A Notes on Agda Notation

We summarize the notation of the Agda language that is necessary to read the code in this thesis.

Names in Agda can contain most characters, including unicode. Unless separated by a space or another reserved character, two subsequent characters are always part of the same name. For example $n{\leq}m$ is a single name, whereas $n \leq m$ is a compound term consisting of three names. The only characters that cannot be part of a name are `.;{}()@"`.

Arguments to Agda definitions can be supplied 'mixfix'. The underscores in a declaration indicate where the arguments go. Underscores in patterns or other binding positions indicate that we do not care about the value and do not bother to give it a name. Most symbols can be part of names: you may assume that unless there is a space between two symbols, they are part of the same name. Casing of names has no formal meaning, although we make a habit of capitalizing types and using lowercase for values.

The name Set is a builtin of Agda and means "type". More precisely, it denotes the universe of types of Level zero. This thesis contains one or two occurrences of the next universe, written Set_1.

Agdaism: An extreme example: *a,b* would be a single name, whereas *a , b* is a pair of two things named *a* and *b* respectively. As you can see, this should be apparent also from the syntax highlighting.

The colors of Agda code are meaningful. Green denotes a function definition, blue a data- or record-type, and pink a constructor. Less frequently, we will see ochre for fields of records, and purple for a module name. Some primitives (like + or Set) are colored black. **Bold black** is used for keywords. *Italics* in code is reserved for parameters.

Function types are written as *telescopes* and permit quantifica-

tion over elements of any particular Set (e.g., $n : \mathbb{N}$), as well as quantification over elements of a universe (e.g., $A : \mathsf{Set}$). For example:

$$\mathsf{repeat} : (A : \mathsf{Set}) \to A \to (n : \mathbb{N}) \to \mathsf{Vec}\ A\ n$$

The function repeat has three arguments: first a type A, then an element of that type, and finally a natural number n. It then returns a vector consisting of n elements of A. As usual, the scope of a name of an argument extends to the end of the type expression, mirroring λ-binding.

When we want the type of an argument x to be inferred, we can write $\forall\ x\ \to\ (...)$. For example, we can omit the type of A in the signature of repeat, so that Agda tries to infer it from the signature of the type constructor Vec:

$$\mathsf{repeat} : \forall\ A \to A \to (n : \mathbb{N}) \to \mathsf{Vec}\ A\ n$$

If instead we want a *value* to be inferred we can write:

$$\{x\ :\ A\}\ \to\ (...).$$

We usually omit these *implicit parameters* from signatures, with the understanding that any undeclared variables in a signature are universally quantified implicit arguments. For example, the signature of append will be written:

$$\mathsf{append} : \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ (n + m)$$

Thus omitting the quantification over the type A and numbers n and m. Implicitly quantified variables are inserted at the front of the type signature in the order that they appear in the signature. Hence, the explicated signature of append is:

$$\mathsf{append} : \forall\ \{A : \mathsf{Set}\}\ \{n\ m : \mathbb{N}\} \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ (n + m)$$

This is accommodated in the literate Agda source of this thesis using Agda's feature for generalizing over declared variable names.

Finally, we can write $\{\{x\ :\ A\}\}\ \to\ (...)$ if we want the argument x to be filled in at the call site by *instance search*. *Instance arguments*[1] generalize type class constraints (as present in, for example, Haskell) and are filled in by the type checker using a search procedure over a set of definitions marked as instances. An instance for A is written:

[1] Devriese et al. 2011. "On the bright side of type classes: instance arguments in Agda"

```
instance a-T : T
a-T = {!!}
```

Functions are inhabited by lambdas, which can be defined inline using the familiar notation for lambda:

```
f : A → B → A
f = λ a b → a
```

Lambdas that are defined by case-distinction on the arguments use the keyword **where**, and we write the different cases on subsequent lines:

```
g : ℕ → A → A
g = λ where
      zero    a → a
      (suc n) a → a
```

In named definitions, the arguments can also be bound on the left of the equality, where pattern matching is always possible:

```
g : ℕ → A → A
g zero    a = a
g (suc n) a = a
```

We can also pattern match on auxiliary values on the left-hand side using **with**-abstraction. For example:

```
h : (n m : ℕ) → ℕ
h n m with n ≟ m
h n .n | yes refl = {!!}
h n m | no proof = {!!}
```

Here we see also why it is relevant that pattern matching on auxiliary values occurs on the left-hand side: in a dependently typed language, case distinction on one value may reveal something about another value. Here, we reveal that m is actually also n. The period indicates that this value is *forced*. That is, dependent pattern matching on auxiliary values may reveal equalities that affect the context and goal types.

A shorter version can be used when a pattern is 'irrefutable':

```
replicate : ∀ n → A → Vec A n
```

```
irrefutable : ℕ
irrefutable with (n :: _) ← replicate 10 1 = n
```

Agda can automatically dismiss the pattern [], such that only a single candidate remains. The notation allows us to write the single pattern on the same line as the scrutinee instead of on a new line.

A shorthand for **with**-abstraction using an equality and matching on refl is to **rewrite** with the equality, like so:

```
h : (n m : ℕ) → n ≡ m → suc n ≡ suc m
h n m eq rewrite eq = {!!}
```

Here, the goal has type suc $m \simeq$ suc m and can be filled with refl.

Pattern matching on empty types is done using the so-called *absurd pattern* (). For example:

```
absurdly : ∀ {A : Set} → 1 ≡ 0 → A
absurdly ()
```

Record types are another important feature that we will make frequent use of. A simple record type $\mathcal{R}$ is declared as:

```
record ℛ : Set where
  constructor mkR
  field
    f₁ : F₁
    f₂ : F₂

  def : F₁ ⊎ F₂
  def = inj₁ f₁
```

The type $\mathcal{R}$ can be understood as tuple with named projections f_1 and f_2. We write $f_1\ r$ to project the field from a value $r : \mathcal{R}$. Fields can be marked as implicit or instance fields with the usual bracketed notation.

The body of the record may contain definitions like def that can use the fields of the record. These definitions take a record value as an extra first parameter on the use-site—i.e., def is used as def r.

Alternatively, record values can be used by opening them, bringing the fields and definitions into scope. For example, assuming again a value $r : \mathcal{R}$ in scope:

```
open ℛ r
myF₁ : F₁
myF₁ = f₁
```

We will not explicitly write such **open** statements, but will occasionally assume it if it is clear from the context with which record value we are working. For example, if we are working with a particular monad that is defined as a record.

Record values can be created using the declared constructor. The constructor takes the (explicit) fields of the record as arguments, and returns a record value. For example, the constructor mkR has type $F_1 \to F_2 \to$ R.

Sometimes we prefer to define record values by *copattern matching*. That is, we define a record value by defining values for all the projections/fields. For example:

```
myℛ : ℛ
f₁ myℛ = {!!}
f₂ myℛ = {!!}
```

Copatterns can be nested if we want to construct a nested record. The above definition by copattern matching is equivalent to the definition using the constructor.

Monadic computations are conveniently expressed using **do**-notation. This notation in Agda is implemented as syntactic sugar and is translated away before type checking. Concretely, we have the following translation from left to right for binding statements:

```
m = do                          m =
  x ← m₁                          m₁ >>= (λ a →
  m₂                              return e)
```

and the following translation for non-binding statements:

```
m = do                          m =
  m₁                              m₁ >> m₂
  m₂
```

The type checker than checks the right-hand side as usual, unbiased towards any particular type or implementation of the operators _>>=_ and _>>_.

B Agda Standard Library Definitions

We make frequent use of Agda's standard library and its notations. Throughout the thesis we stick to the names used in the standard library. The precise version is documented with the Agda sources accompanying the thesis.

We explain all but the most standard definitions within the thesis. In this appendix we define these most basic types. Just like in the rest of the thesis we slightly simplify definitions by omitting universe polymorphism. The main purpose of this section is as a reference for all the names of the constructors and projections of these types.

The empty type $\bot$, unit type $\top$, natural numbers $\mathbb{N}$, booleans Bool, lists List A, and optional values of A Maybe A are defined as follows:

```
data ⊥ : Set where
  -- no constructors

record ⊤ : Set where
  constructor tt
  -- no fields

data ℕ : Set where
  zero : ℕ
  suc  : (n : ℕ) → ℕ

data Bool : Set where
  true false : Bool

data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A

[_] : A → List A
[ a ] = a :: []
```

```
data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A
```

Dependent pairs $\Sigma\ A\ B$ are defined as a record, with some syntactic sugar for non-dependent pairs $A \times B$:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁

_×_ : Set → Set → Set
A × B = Σ A (λ _ → B)
```

Something that frequently confuses people is the notation in Agda for (proof-relevant) existential quantification. In this thesis we will write it as $\exists\langle\ P\ \rangle$. This should be read as $\exists\langle\_\rangle$ existentially quantifying over all n parameters of $P\ :\ X_1 \to ... \to X_n \to$ Set. The value of this type is a (n+1)-tuple represented as a right-nested dependent product. That is, if $P\ :\ A \to B \to$ Set, then $\exists\langle\ P\ \rangle = \Sigma\ A\ (\lambda\ a \to \Sigma\ B\ (\lambda b \to P\ a\ b))$. For non-dependent products $\Sigma\ A\ (\lambda\ \_ \to B)$ we write $A \times B$. Because existential quantification is defined in terms of dependent pairs, we destruct it by pattern matching on the constructor _,_, or by projecting one of the fields of the type Σ.

Propositional equality _≡_ is defined in the usual way between two values of any type A:

```
data _≡_ {A : Set} : A → A → Set where
  refl : ∀ {a} → a ≡ a
```

Negation is defined as implying the empty type:

```
¬_ : Set → Set
¬ A = A → ⊥

_≢_ : A → A → Set
a ≢ b = ¬ (a ≡ b)
```

Summary

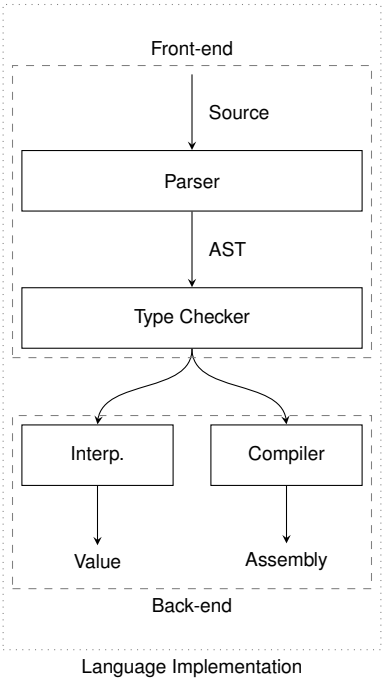
Programming language implementations bridge the gap between what the program developer sees and understands, and what the computer executes. Hence, it is crucial for the reliability of software that language implementations are correct.

Correctness of an implementation is judged with respect to a criterion. In this thesis, we focus on the criterion *type correctness*, striking a balance between the difficulty of the assessment of the criterion and its usefulness to rule out errors throughout a programming language implementation. If both the front- and the back-end fulfill their role in maintaining the type contract between the programmer and the language implementation, then unexpected type errors will not occur when the program is executed.

To verify type correctness throughout a language implementation, we want to establish it formally. That is, we aim to give a specification of program typing in a formal language, and to give a mathematical proof that every part of the language implementation satisfies the necessary property to make the whole implementation type-correct.

Type checkers ought to be *sound* and only accept programs that are indeed typeable according to the specification of the language. Interpreters should be *type safe*, and reduce expressions to values of the same type. Program compilers should *preserve well-typing* when they transform programs. These properties are essential for implementations of typed programming languages, ensuring that the typing of the source program is a meaningful notion that can be trusted by the programmer to prevent certain errors from occurring during program execution.

A conventional formal type-correctness result consists of separate specification, implementation, and proof. The separation makes it difficult to construct language implementations such that their correctness can be formally verified. It is also difficult to relate a



specification to a given implementation if the latter is not developed specifically for the purpose of verification. This in turn makes a formal proof at present infeasible unless the specification, implementation, and proof are developed in tandem.

To address these issues and to make it more feasible and attractive to develop formally verified language implementations, this thesis pursues a more integrated vision to verification: language specifications and/or implementations ought to be written in meta-languages that ensure type-correctness and avoid the need for a separate proof.

This thesis considers several methods to develop formally (and mechanically) verified language implementations.

For language front-ends, this thesis investigates an approach to automatically obtain verified type checkers from declarative specifications of type systems. This specification is given by the language developer as typing rules in the Statix meta-language. An important aspect of the Statix language are its primitive for specifying (global) static binding, simplifying the specification of, for example, module systems, and name resolution of class members in object-oriented languages like Java. This thesis reduces Statix to a small formal language Statix-core. We give a novel, compact declarative semantics for Statix-core that gives meaning to typing rules in relation to a given structured symboltable in the form of a scope graph. We also give Statix-core an operational semantics that can determine whether a given source program has a scope graph that makes it well-typed. This effectively delivers a type checker for the specified language.

To show that such type checkers are type-correct, we give a formal proof of a soundness theorem for the operational semantics, relative to the formal meaning of the typing rules given by the declarative semantics. We also prove that the non-deterministic operational semantics is confluent. The key contribution of this thesis that makes the proof possible is the idea that we can reason about Statix specifications using separation logic. While separation logic is typically used to reason about memory usage, we use it to reason about unique declarations in the structured symbol table—a *scope graph*—belonging to a program.

For language back-ends, this thesis investigates techniques for implementing intrinsically typed definitional interpreters and compilers. The idea is to use a dependently typed functional language,

like Agda, to integrate the specification of well-typing in the representation of the program that is being interpreted or transformed. This enables the integration of the type-correctness into the language implementation. Hence, Agda fulfills both the role of the programming language wherein one writes the implementation, and the role of the meta-language wherein we specify and prove the correctness. In previous work this technique has been applied to simply typed languages with great success: intrinsic typing avoids all the overhead of an external proof. It also simplifies the implementation itself, because one avoids the need to manually handle type errors that are prohibited by the integrate type correctness theorem.

This thesis makes several technical contributions to scale this approach from the simply typed languages to languages with mutable references à la ML, a concurrent functional language with linear types and session-typed communication, and a low-level bytecode language with labels and jumps. We accomplish this by delivering functional abstractions that not only encapsulate computational work, but also *proof work*. We develop these functional abstractions on top of logical languages that we embed in Agda, inspired by more semantic approaches to proving type-correctness. Specifically, we develop an embedding of proof-relevant monotone predicates for the invariant of references, and we develop proof-relevant separation logic to abstract over the sub-structural invariants of the interpreter for the linear functional language, and also of the compiler outputting bytecode. We deliver interpreters and a compiler that are total and have almost no proof overhead. Both the logical languages and the functional abstractions that we develop on top of it are reusable, because they abstract over—among others—the algebras that specify the invariants.

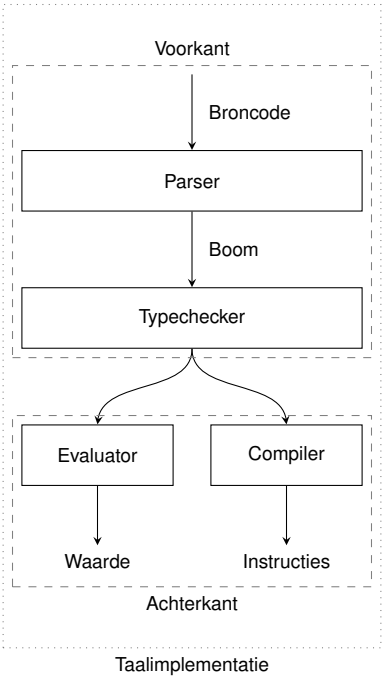
Samenvatting

Implementaties van programmeertalen overbruggen het gat tussen hetgeen de programmeur te zien krijgt, en hetgeen de computer uitvoert. Het is daarom cruciaal voor de betrouwbaarheid van software dat deze implementaties juist zijn.

De juistheid van een implementatie wordt relatief aan een criterium beoordeeld. In dit proefschrift focussen we op het criterium type-juistheid. Zo balanceren we de moeilijkheidsgraad van de evaluatie van het criterium en de bruikbaarheid om fouten uit te sluiten uit de gehele programmeertaal-implementatie. Als zowel de voorals de achterkant van de implementatie hun rol vervullen in het onderhouden van type-juistheid, dan kunnen onverwachte typefouten niet voorkomen tijdens het uitvoeren van een programma.

Om type-juistheid in de gehele implementatie van een programmeertaal te waarborgen willen we het formeel verifiëren. Dat betekent dat we pogen om een specificatie te geven van programmatypering in een formele taal, en om een wiskundig bewijs te geven dat elk onderdeel van de taal-implementatie de benodigde eigenschappen bezit om het geheel type-juist te maken. Typecheckers behoren aan een degelijkheidsstelling te voldoen, en moeten alleen programmas accepteren die typeerbaar zijn volgens de typeregels van de taalspecificatie. Programma-evaluators moeten typeveilig zijn, en expressies reduceren naar waarden met hetzelfde type. Programma-compilers behoren getypeerdheid te behouden wanneer ze programmas transformeren. Deze eigenschappen zijn essentieel om te garanderen dat de getypeerdheid van het bronprogramma een betekenisvolle notie is waar de programmeur op kan vertrouwen om bepaalde fouten tijdens programma evaluatie uit te sluiten.

Een conventioneel bewijs van een type-juistheidsresultaat bestaat uit gescheiden specificatie, implementatie, en bewijsvoering. Deze scheiding maakt het moeilijk om taal-implementaties te ontwikkelen die formeel bewijsbaar juist zijn. Het is ook moeilijk om een



type-specificatie te relateren aan een implementatie als deze laatste niet specifiek ter verificatie is ontwikkeld. Dit maakt het op zijn beurt onmogelijk om ten hedendage een formeel bewijs van juistheid te geven tenzij de specificatie, de implementatie, en het bewijs gelijktijdig worden ontwikkeld.

Om deze problemen aan te pakken en om het meer doenlijk en aantrekkelijker te maken om formeel geverifieerde taal-implementaties te ontwikkelen, streeft dit proefschrift een meer geïntegreerde visie tot verificatie na: taal specificaties en/of implementaties zouden moeten worden geschreven in meta-talen die type-juistheid garanderen en de noodzaak tot een extern bewijs vermijden.

Dit proefschrift beschouwt verschillende methoden om formeel (en mechanisch) geverifieerde taal-implementaties te ontwikkelen.

Voor de voorkant van de implementatie onderzoekt dit proefschrift een methode om automatisch geverifieerde type-checkers te verkrijgen uit declaratieve specificaties van type-systemen. Deze specificatie wordt door de taalontwikkelaar gegeven in de Statix meta-taal in de vorm van typeregels. Een belangrijk aspect van Statix zijn de primitieven voor het specificeren van (globale) statische binding om de specificatie van bijvoorbeeld module systemen en de herleiding van namen op objecten in talen zoals Java gemakkelijker te maken. Dit proefschrift reduceert Statix tot een kleine formele taal Statix-core. We geven een vernieuwde compacte declaratieve semantiek voor Statix-core die de betekenis van typeregels geeft in relatie tot een gegeven gestructureerde symbooltabel in de vorm van een scopegraaf. Vervolgens geven we ook een operationele semantiek voor Statix-core welke bepaalt of een gegeven bronprogramma een passende scopegraaf heeft die het programma goed getypeerd maakt. Daarmee hebben we effectief een type-checker voor de gespecificeerde taal verkregen.

Om te laten zien dat deze type-checkers type-juist zijn, geven we een formeel bewijs van een degelijkheidsstelling voor de operationele semantiek, met betrekking tot de formele betekenis van de typeregels gegeven door de declaratieve semantiek. Ook bewijzen we dat de non-deterministische operationele semantiek samenvloeiend (in het Engels 'confluent') is. De sleutel contributie van dit proefschrift die de bewijsvoering mogelijk maakt, is het idee dat we kunnen redeneren over Statix specificaties middels separatie-logica. Terwijl separatie-logica typisch gebruikt wordt om te redeneren over het ge-

bruik van geheugen, gebruiken wij het om te redeneren over unieke declaraties in een scopegraaf.

Voor de achterkant van taal-implementaties onderzoekt dit proefschrift technieken voor de implementatie van intrinsiek-getypeerde definitionele evaluators en compilers. Het idee hierbij is om gebruik te maken van een functionele programmeertaal met types die waarde-afhankelijk kunnen zijn—zoals Agda—om de specificatie van type-juistheid te integreren in de representatie van het programma dat door de implementatie wordt geëvalueerd of getransfomeerd. Dit maakt het mogelijk om het juistheidsbewijs te integreren in de taalimplementatie. Hierbij vervult Agda dus zowel de rol van de programmeertaal waarin de implementatie wordt geschreven, als de meta-taal waarin de specificatie en bewijsvoering plaatsvind. In voorgaand werk is deze techniek met groot succes toegepast op talen met simpele types: intrinsiek-getypeerde definities weten al het werk van een extern bewijs te vermijden. De techniek vereenvoudigt ook de implementatie zelf, omdat mogelijke typefouten die op deze manier gelijk door het correctheidsbewijs worden uitgesloten niet langer hoeven worden afgehandeld.

Dit proefschrift maakt een aantal technische contributies om de techniek te schalen van talen met simpele types naar talen met referenties à la ML, een functionele taal met lineaire types en getypeerde communicatie tussen samenwerkende processen, en een laag-niveau instructie taal met labels en goto instructies. We maken dit mogelijk door functionele abstracties te ontwikkelen die niet alleen computationeel werk, maar ook bewijslast omsluiten. We ontwikkelen deze abstracties bovenop logische talen die we inbedden in Agda, geïnspireerd door meer semantische methoden voor het bewijzen van type-juistheid. Voor de invarianten van referenties ontwikkelen we bewijsrelevante monotone predicaten. Voor de sub-structurele invarianten van de evaluator voor de lineaire functionele taal en van de compiler naar bytecode, ontwikkelen we een bewijsrelevante separatie-logica. De resulterende evaluators en compiler zijn totaal en zijn bijna vrij van bewijslast. Zowel de logische talen als de daarbovenop ontwikkelde functionele abstracties zijn herbruikbaar doordat we abstraheren over—onder andere—de algebras die de invarianten beschrijven.

Titles in the IPA Dissertation Series since 2018

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems*. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages*. Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities*. Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis*. Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing*. Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

D.Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VUA. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A.J. Rouvoet. *Correct by Construction Language Implementations*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-06

Curriculum Vitae

1991	Born on September 4th, Nunspeet, The Netherlands.
1994–2003	Elementary school, R. de Jagerschool, Woerden, The Netherlands.
2003–2009	High School (VWO), Kalsbeek College, Woerden, The Netherlands.
2009–2013	BSc in Electrical Engineering and BSc in Computer Science, Delft University of Technology, Delft, The Netherlands. Graduated with cum laude distinction.
2013–2016	MSc in Computer Science, Delft University of Technology, Delft, The Netherlands. Thesis title: "Programs for Free: Towards the Formalization of Implicit Resolution in Scala", supervised by Dr.ir. Sandro Stucki and Prof.dr. Erik Meijer.
2015	Software engineering intern at @WalmartLabs, San Bruno, California, The United States of America.
2012–2016	Software engineer at Occator, Den Hoorn, The Netherlands.
2016–2017	Medior software engineer at Science & Technology, Delft, The Netherlands.
2017–2021	PhD candidate in Computer Science, Delft University of Technology, Delft, The Netherlands. Thesis title: "Correct by Construction Language Implementations", supervised by Prof.Dr. Eelco Visser and Dr.ir. Robbert Krebbers.
2018	Oregon Programming Languages Summerschool (OPLSS), Eugene, Oregon, The United States of America.
2019	EUTYPES summerschool, Ohrid, North Macedonia.
2020	Scottish Summer School on Programming Languages and Verification (SPLV), online due to COVID-19.
2021–	Post-doctoral researcher, Delft University of Technology, Delft, The Netherlands.