

A Modal Analysis of Staged Computation

ROWAN DAVIES AND FRANK PFENNING

Carnegie Mellon University, Pittsburgh, Pennsylvania

Abstract. We show that a type system based on the intuitionistic modal logic S4 provides an expressive framework for specifying and analyzing computation stages in the context of typed λ -calculi and functional languages. We directly demonstrate the sense in which our $\lambda_{\epsilon}^{\rightarrow\Box}$ -calculus captures staging, and also give a conservative embedding of Nielson and Nielson's two-level functional language in our functional language Mini-ML $^{\Box}$, thus proving that binding-time correctness is equivalent to modal correctness on this fragment. In addition, Mini-ML $^{\Box}$ can also express immediate evaluation and sharing of code across multiple stages, thus supporting run-time code generation as well as partial evaluation.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*partial evaluation*; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*type structure*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—*lambda calculus and related systems, modal logic, proof theory*

General Terms: Languages, Theory

Additional Key Words and Phrases: Binding times, run-time code generation, staged computation

1. Introduction

Dividing a computation into separate stages is a common informal technique for the derivation of algorithms [Jørring and Scherlis 1986]. For example, instead of directly matching strings against a regular expression we may first compile the regular expression into a finite automaton and then execute the same automaton on different strings. Because significant efficiency gains can often be realized, there is a substantial body of work concerned with the automation of staged computation. *Partial evaluation* (see, e.g., Jones et al. [1993]) divides the computation into two stages based on the early availability of some function arguments.

This is an extended and revised version of the conference paper [Davies and Pfenning 1996].

This work was sponsored in part by the National Science Foundation under grant CCR-9619832 and by the Advanced Research Projects Agency (ARPA) under Order No. 8313.

R. Davies was partly supported by a Hackett Studentship from the University of Western Australia. Part of this work was completed during a visit by R. Davies to BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).

Authors' address: 5000 Forbes Ave., Pittsburgh, PA 15213-3891.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery (ACM), Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 2001 ACM 0004-5411/01/0500-0555 \$05.00

In practice this appears most successful when supported by *binding-time analysis* [Gomard and Jones 1991], which statically determines which parts of a computation may be carried out in the first phase, and which parts remain to be done in the second phase.

It often takes considerable ingenuity to write programs in such a way that they exhibit proper binding-time separation, that is, that the computation intended to occur when the early arguments become available can in fact be carried out. From a programmer's point of view it is therefore desirable to declare the expected binding-time separation and obtain constructive feedback when the computation may not be staged as expected. This suggests that the binding-time properties of a function should be expressed in a prescriptive type system, and that binding-time analysis should be a form of type checking. The work on two-level functional languages [Nielson and Nielson 1992] and some work on partial evaluation (e.g., [Gomard and Jones 1991]) shows that this view is indeed possible.

Up to now, these type systems have been motivated *algorithmically*, that is, they are explicitly designed to support specialization of a function to its early arguments. In this paper we show that they can also be motivated logically, and that the proper logical system for expressing computation stages is the intuitionistic variant of the modal logic S4 [Prawitz 1965]. This observation immediately gives rise to a natural generalization of standard binding-time analysis by allowing multiple computation stages, initiation of successor stages, and sharing of code across multiple stages. Such extensions are normally considered external issues. For example, Jones [1992] describes a typed framework for such concepts, but only at the level of operations on whole programs. Our framework instead provides these operations *within* the language of programs. This makes our approach particularly relevant to run-time code generation, where specialization takes place when the program is executed. Indeed, the authors and others have designed and implemented an extension of ML based on the type system described here which generates and executes abstract machine code at run time [Wickline et al. 1998a; 1998b].

One of our conclusions is that when we extend the Curry–Howard isomorphism between proofs and programs from intuitionistic logic to the intuitionistic modal logic S4 we obtain a natural and logical explanation of computation stages. The isomorphism relates proofs in modal logic to functional programs which manipulate program fragments for later stages. Each world in the Kripke semantics of modal logic corresponds to a stage in the computation, and a term of type $\Box A$ corresponds to code to be executed in a future stage of the computation. The modal restrictions imposed on terms of type $\Box A$ guarantee that a function of type $B \rightarrow \Box A$ can carry out all computation concerned with its argument while generating the residual code of type A .

We begin by considering $\lambda_e^{\rightarrow\Box}$, a modal λ -calculus based on a natural-deduction formulation of intuitionistic modal S4. The presentation is new, but draws on ideas in Bierman and de Paiva [1996], Pfenning and Wong [1995], and Girard [1993].

We then construct a functional language Mini-ML $^\Box_e$ by augmenting $\lambda_e^{\rightarrow\Box}$ with a fixpoint operator, natural numbers, and pairs and endow it with a natural call-by-value operational semantics along the lines of Mini-ML [Clément et al. 1986].

Mini-ML $^\square_e$ can be somewhat awkward because it often requires a broad syntactic structuring of the program to directly reflect staging. This simplifies the study of staging properties of Mini-ML $^\square_e$, but it also makes it difficult to directly relate it to previous work on staged languages, such as two-level languages [Nielson and Nielson 1992]. We thus consider a more implicit formulation of S4 motivated by its Kripke semantics following Martini and Masini [1996] and Pfenning and Wong [1995], and then augment it as before to form Mini-ML $^\square$. With some syntactic sugar, Mini-ML $^\square$ is intended to serve as the basis for a conservative extension of ML with practical means to express and check staging of computation. The operational semantics of Mini-ML $^\square$ is given by a type-preserving translation to Mini-ML $^\square_e$ whose correctness is not entirely trivial. This translation also describes the first phase of a plausible compilation strategy for Mini-ML $^\square$ for run-time code generation.

We then exhibit a simple full and faithful embedding of Nielson & Nielson's two-level language [Nielson and Nielson 1992] in Mini-ML $^\square$, providing further evidence that Mini-ML $^\square$ provides an intuitively appealing, technically correct, and logically motivated view of staged computation.

2. A Modal λ -Calculus

In this section, we present the modal λ -calculus $\lambda_e^{\rightarrow \square}$. We start by directly motivating the calculus in terms of “manipulation of code” and relate this to modal logic. We then present typing rules based on a natural deduction system for modal S4, give β and η rules for the modal \square operator, and show that they satisfy subject reduction and expansion, respectively. We also demonstrate the relationship between $\lambda_e^{\rightarrow \square}$ and computation staging via two theorems. We assume some familiarity with logical concepts such as modal logic, natural deduction, and the Curry–Howard isomorphism. We refer the reader to some of our recent work [Pfenning and Davies 2001] for a more basic discussion of these concepts for and to Chellas [1980] for an introduction to modal logic.

2.1. NATURAL DEDUCTION FOR VALIDITY. A common feature of many forms of staged computation is the manipulation of code. Macro expanders and partial evaluators typically manipulate source expressions, run-time code generators typically manipulate object code or some form of intermediate code. To show how such manipulation of code may be accounted for in a typed framework, we start with a typed λ -calculus and introduce a new type constructor \square , where $\square A$ represents *code of type A*. This type remains abstract in the sense that we do not commit ourselves to a particular way of implementing it. In this way, our type system can support diverse applications.

Next we have to decide which operations should be supported on code. First, we should be able to manipulate an arbitrary *closed* expression as code. This suggests a constructor **box** where **box** $E : \square A$ if $E : A$ in the empty context. This is essentially the modal rule of necessitation. The second means of constructing code is by *substitution*: we can substitute code for a free variable appearing in code to obtain code. In a meaningful type system, such substitution must be “hygienic” and rename bound variables if necessary to avoid capture. The restriction that we can only substitute *code* (and not arbitrary expressions) into code is reflected exactly in one of Prawitz's variants of the modal necessitation

rule [Prawitz 1965]: we can infer that $\mathbf{box} E : \Box A$ from $E : A$ if all hypotheses of the latter derivation are of the form $x : \Box B$. This means that every free variable x in E must have a type of the form $\Box B$. Prawitz's elimination rule allows us to infer A from $\Box A$. In terms of the functional interpretation, this suggests *evaluation*: we execute the *code* of type A to obtain a *value* of type A .

Unfortunately, the natural deduction formulation of modal logic based on these two rules does not obey subject reduction, as noticed by Prawitz himself and demonstrated by the following derivation:

$$\frac{\frac{\frac{}{\Box A \vdash \Box A}}{B, B \rightarrow \Box A, \Box A \vdash \Box \Box A} \quad \frac{}{B, B \rightarrow \Box A \vdash B \rightarrow \Box A} \quad \frac{}{B, B \rightarrow \Box A \vdash B}}{B, B \rightarrow \Box A \vdash \Box \Box A} \quad \frac{}{B, B \rightarrow \Box A \vdash \Box A}}{B, B \rightarrow \Box A \vdash \Box \Box A}$$

This derivation contains a redex at the outermost level, due to the introduction and then elimination of \rightarrow . Attempting to reduce this redex yields the following, which violates the restriction on Prawitz's necessitation rule:

$$\frac{\frac{}{B, B \rightarrow \Box A \vdash B \rightarrow \Box A} \quad \frac{}{B, B \rightarrow \Box A \vdash B}}{B, B \rightarrow \Box A \vdash \Box A} \quad \frac{}{B, B \rightarrow \Box A \vdash \Box \Box A}$$

We can trace the difficulty to the global side-condition on the necessitation rule which requires assumptions to be of a particular form. If we express this condition directly on the level of the judgments, we are led to a different system which does satisfy subject reduction and other properties desirable for a system of natural deduction. To this end, we introduce two basic judgments on propositions, “ A is true” and “ A is valid.” We have hypotheses expressing that certain propositions are true and others are valid. We write

$$(A_1, \dots, A_m); (B_1, \dots, B_n) \vdash^e C$$

to express

Under the hypothesis that A_1, \dots, A_m are valid and B_1, \dots, B_n are true, C is true.

Since our main goal is the analysis of the Curry–Howard isomorphism between proofs and programs, we label the hypotheses and annotate C with a proof term E .

$$(u_1 : A_1, \dots, u_m : A_m); (x_1 : B_1, \dots, x_n : B_n) \vdash^e E : C$$

Here and throughout this paper, we presuppose that all variables labelling hypotheses are distinct.

Taking the functional view for a moment, we think of u_1, \dots, u_m as variables ranging over code and x_1, \dots, x_n as variables ranging over values that may occur free in the expression E . Generally, we write Δ for a context $u_1 : A_1, \dots, u_m : A_m$ declaring *modal variables* u (also called *code variables*) and Γ for a context $x_1 : B_1, \dots, x_n : B_n$ declaring *ordinary variables* x (also called *value variables*).

But how do we conclude that A is valid? In informal terms, A is valid if it is true under all possible interpretations. In other words, its derivation may not depend on any hypotheses about the truth of propositions. That is, we judge that C is valid under the hypothesis that A_1, \dots, A_m are valid if

$$(A_1, \dots, A_m); \cdot \vdash^e C$$

or, with proof terms,

$$(u_1:A_1, \dots, u_m:A_m); \cdot \vdash^e E:C$$

With respect to our functional interpretation, this means that E contains only free code variables, but no free value variables.

We now develop the inference rules characterizing the judgments and then introduce the logical connectives. First, we have

$$\frac{x:A \text{ in } \Gamma}{\Delta; \Gamma \vdash^e x:A} \text{ovar}$$

since we can conclude that A is true from the hypothesis that A is true. But it is certainly also the case that A is true if A is valid.

$$\frac{u:A \text{ in } \Delta}{\Delta; \Gamma \vdash^e u:A} \text{mvar}$$

The transition from a judgment of validity to that of truth corresponds on the functional side to a transition from code to value. We will use this later to encode evaluation.

Second, we consider the substitution principles which are derived from the nature of the hypothetical judgments. In purely logical terms: if we have a derivation showing that C is true from the hypothesis that A is true, then we can substitute an actual derivation establishing the truth of A for all uses of the hypothesis. This results in a derivation for the truth of C , which no longer depends on the hypothesis. With proof terms, the substitution principle for ordinary hypotheses reads:

Ordinary Substitution Principle

If $\Delta; \Gamma \vdash^e E_1:A$ and $\Delta; (\Gamma, x:A, \Gamma') \vdash^e E_2:B$
 then $\Delta; (\Gamma, \Gamma') \vdash^e [E_1/x]E_2:B$.

Similarly, we should be able to substitute a derivation demonstrating the validity of A for all uses of the hypothesis that A is valid.

Modal Substitution Principle

If $\Delta; \cdot \vdash^e E_1:A$ and $(\Delta, u:A, \Delta'); \Gamma \vdash^e E_2:B$
 then $(\Delta, \Delta'); \Gamma \vdash^e [E_1/u]E_2:B$.

It is critical here that A is valid and not just true, which should be obvious from what is said above. Therefore, we must require $\Delta; \cdot \vdash^e E_1:A$ rather than just $\Delta; \Gamma \vdash^e E_1:A$ (which would be unsound).

Eventually, when our system is complete, we have to *prove* the validity of the two substitution principles to verify that there is no mistake in the design of our rules. Similar guiding properties of hypothetical judgments are *exchange* (the order of hypotheses should not matter), *weakening* (hypotheses need not be used) and *contraction* (hypotheses may be used more than once). All of these are proved in Section 2.4.

The next step is to introduce the logical connectives and operators. In natural deduction, these are characterized by introduction and elimination rules which must match in an appropriate way. One of the underlying principles of natural deduction is that connectives should be orthogonal to each other: each introduction or elimination rule should refer only to the connective whose meaning we define.

We first discuss this using the familiar implication (or function type, under the Curry–Howard correspondence). We want to express that $A \rightarrow B$ should be true if B is true under the hypothesis that A is true.

$$\frac{\Delta; (\Gamma, x:A) \vdash^e E:B}{\Delta; \Gamma \vdash^e \lambda x:A. E:A \rightarrow B} \rightarrow I.$$

Note that Δ is not affected—validity does not enter the considerations for this connective. On proof terms, this rule explicitly introduces the function that maps proofs of A to proofs of B .

Conversely, if we know that $A \rightarrow B$ is true then B should be true under hypothesis A . So if we also know that A is true, we can conclude that B must be true.

$$\frac{\Delta; \Gamma \vdash^e E_2:A \rightarrow B \quad \Delta; \Gamma \vdash^e E_1:A}{\Delta; \Gamma \vdash^e E_2 E_1:B} \rightarrow E.$$

On proof terms, this applies the function E_2 , which maps proofs of A to proofs of B to the given proof E_1 of A .

How do we know the introduction and elimination rules match and thus define a meaningful connective? We should verify two conditions: *local soundness* and *local completeness*. Local soundness ensures that we cannot gain information by introducing a connective and then immediately eliminating it—we must already be able to make the same judgment without the detour. This guarantees that the elimination rules are not too strong. Local completeness ensures that we can recover all information present in a connective: there is some way to apply the elimination rules so we can reconstitute a proof of the original proposition using its introduction rules. This guarantees that the elimination rules are not too weak.

On proof terms, local soundness and completeness are witnessed by local reduction and expansion, taking advantage of the substitution principles.

$$\frac{\frac{\Delta; \Gamma, x:A \vdash^e E_2:B}{\Delta; \Gamma \vdash^e (\lambda x:A. E_2):A \rightarrow B} \rightarrow I \quad \frac{\Delta; \Gamma \vdash^e E_1:A}{\Delta; \Gamma \vdash^e (\lambda x:A. E_2) E_1:B} \rightarrow E}{\Delta; \Gamma \vdash^e (\lambda x:A. E_2) E_1:B} \Rightarrow \Delta; \Gamma \vdash^e [E_1/x]E_2:B$$

Here, \mathcal{D}'_2 is constructed by substitution of \mathcal{D}_1 into \mathcal{D}_2 , as indicated in the discussion of the (ordinary) substitution principle. On proof terms, we have ordinary β -reduction. Local completeness is witnessed by η -expansion.

$$\Delta; \Gamma \vdash^e E : A \rightarrow B \Rightarrow \frac{\frac{\Delta; \Gamma, x:A \vdash^e E : A \rightarrow B \quad \frac{}{\Delta; \Gamma, x:A \vdash^e x : A} \text{ovar}}{\Delta; \Gamma, x:A \vdash^e E x : B} \rightarrow E}{\Delta; \Gamma \vdash^e (\lambda x:A. E x) : A \rightarrow B} \rightarrow I$$

Here, \mathcal{D}' is constructed by weakening from \mathcal{D} (we add the unused hypothesis $x:A$), which has no effect on the proof term E . On proof terms, therefore, we have ordinary η -expansion.

Next we consider the modal operator. $\Box A$ should be true if A is valid. Written as an inference rule:

$$\frac{\Delta; \cdot \vdash^e E : A}{\Delta; \Gamma \vdash^e \mathbf{box} E : \Box A} \Box I.$$

Note how the premise enforces that A is valid by requiring the ordinary context to be empty. On proof terms, this means that only modal variables from Δ can occur free in E .

The corresponding elimination is not straightforward. For example, Prawitz's rule from above which concludes A from $\Box A$ is locally sound but not complete. Intuitively, this should be clear because we are losing information when we make the step from " $\Box A$ is true" to " A is true." An alternative rule that concludes " A is valid" from " $\Box A$ is true" is unsound, because the judgment that $\Box A$ is true may actually depend on hypotheses about the truth of other propositions.

Instead we reason as follows: if $\Box A$ is true under some hypotheses, then any judgment we make under the additional hypothesis that A is valid, must in fact be evident.

$$\frac{\Delta; \Gamma \vdash^e E_1 : \Box A \quad (\Delta, u:A); \Gamma \vdash^e E_2 : B}{\Delta; \Gamma \vdash^e \mathbf{let} \mathbf{box} u = E_1 \mathbf{in} E_2 : B} \Box E.$$

Thus, the elimination rule for \Box introduces a modal hypothesis and the corresponding term construct has the form of a **let**. From the functional point of view, E_1 represents a value of type $\Box A$ containing some code. This code is accessible in E_2 with the name u . Local soundness and completeness with this construction are easily verified. Local soundness is guaranteed by the reduction

$$\frac{\frac{\frac{\Delta; \cdot \vdash^e E_1 : A}{\Delta; \Gamma \vdash^e \mathbf{box} E_1 : \Box A} \Box I \quad (\Delta, u:A); \Gamma \vdash^e E_2 : B}{\Delta; \Gamma \vdash^e \mathbf{let} \mathbf{box} u = \mathbf{box} E_1 \mathbf{in} E_2 : B} \Box E}{\Delta; \Gamma \vdash^e [E_1/u] E_2 : B} \Rightarrow \Delta; \Gamma \vdash^e [E_1/u] E_2 : B$$

where \mathcal{D}'_2 is the derivation constructed by substitution as indicated in the modal substitution principle.

The expansion below demonstrates local completeness, since the result of the elimination rule applied to a derivation of $\Box A$ contains enough information to reconstitute a derivation of $\Box A$.

$$\frac{\Delta; \Gamma \Vdash^{\mathcal{D}} E : \Box A \quad \frac{\frac{}{(\Delta, u:A); \cdot \Vdash^e u : A} \text{mvar} \quad \frac{}{(\Delta, u:A); \Gamma \Vdash^e \mathbf{box} u : \Box A} \Box I}{\Delta; \Gamma \Vdash^e \mathbf{let} \mathbf{box} u = E \mathbf{in} \mathbf{box} u : \Box A} \Box E$$

Other standard logical connectives such as negation, conjunction, disjunction, universal and existential quantification can be defined by introduction and elimination rules in a similar manner to implication—they do not need to directly interact with the modal hypotheses. Since we are in the intuitionistic setting, the modal possibility operator $\Diamond A$ cannot be defined via negation. It can be characterized directly by introduction and elimination rules which are locally sound and complete, but only if we introduce a new judgment “ A is possibly true,” as we demonstrate in Pfenning and Davies [2001]. We do not show the details here since it does not directly concern our main objective.

Our presentation simplifies that of the modal λ -calculus $\lambda_e^{\rightarrow\Box}$ from Bierman and de Paiva [1996] and Pfenning and Wong [1995] by eliminating the need for simultaneous substitution while preserving subject reduction. It is inspired by sequent calculi proposed by Andreoli [1992] for linear logic and by Girard [1993] for LU. Wadler [1993] has formulated a linear λ -calculus with two contexts, which shares some features with our calculus. The methodology we followed is based on that of Martin-Löf [1996], and is described in more detail in Pfenning and Davies [2001]. One novel feature of our methodology is normative use of local soundness and completeness as witnessed by β -reduction and η -expansion. Note that only β -reduction has computational significance, while η -expansion internalizes an extensionality principle.

The elimination construct for \Box allows us to bind a variable u in Δ to code of type A , written as **let box** $u = E_1$ **in** E_2 . Evaluation of code, certainly one of the most fundamental operations, is then definable by

$$eval \equiv (\lambda x : \Box A. \mathbf{let} \mathbf{box} u = x \mathbf{in} u) : \Box A \rightarrow A.$$

Here, and from now on, we associate \Box more strongly than \rightarrow to avoid excessive parentheses. Note that the opposite coercion, $\lambda x : A. \mathbf{box} x$, cannot be well-typed, since x is an arbitrary argument and will not necessarily be bound to code. Furthermore, it violates the concept of stage separation since x is an “early” argument which we refer to “late,” that is, inside **box**. Here are two other examples of modal propositions and proofs from which the natural deductions can be easily reconstructed.

$$\vdash^e \lambda x : \Box(A \rightarrow B). \lambda y : \Box A. \mathbf{let} \mathbf{box} u = x \mathbf{in} \mathbf{let} \mathbf{box} v = y \mathbf{in} \mathbf{box} (uv)$$

$$: \Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$$

$$\vdash^e \lambda x : \Box A. \mathbf{let} \mathbf{box} u = x \mathbf{in} \mathbf{box} \mathbf{box} u$$

$$: \Box A \rightarrow \Box \Box A$$

Note that the first law holds in all modal logics, while the second and *eval* above correspond to reflexivity and transitivity of the accessibility relation between worlds in the Kripke [1963] semantics in axiomatic formulations of modal logics.

2.2. SYNTAX. We now summarize the system of natural deduction for S4 and its properties.

Types	$A ::= a \mid A_1 \rightarrow A_2 \mid \Box A$
Terms	$E ::= x \mid \lambda x:A. E \mid E_1 E_2 \mid$ $u \mid \mathbf{box} E \mid \mathbf{let} \mathbf{box} u = E_1 \mathbf{in} E_2$
Ordinary Contexts	$\Gamma ::= \cdot \mid \Gamma, x:A$
Modal Contexts	$\Delta ::= \cdot \mid \Delta, u:A$

We use A, B for types, a for base types, x for ordinary variables and u for modal variables assuming that any variable can be declared at most once in a context. Bound variables may be renamed tacitly, and leading \cdot 's may be omitted from contexts. We write $[E'/x]E$ (and similarly for modal variables) for the result of substituting E' for x in E , renaming bound variables as necessary in order to avoid the capture of free variables in E' .

2.3. TYPING RULES

$\Delta; \Gamma \vdash^e E : A$ E has type A in modal context Δ and ordinary context Γ .

Our system has the property that a valid term has a unique type and typing derivation, except for possibly unused hypotheses.

λ -calculus Fragment

$$\begin{array}{c}
 \frac{x:A \text{ in } \Gamma}{\Delta; \Gamma \vdash^e x : A} \text{ovar} \\
 \\
 \frac{\Delta; (\Gamma, x:A) \vdash^e E : B}{\Delta; \Gamma \vdash^e \lambda x:A. E : A \rightarrow B} \rightarrow I \\
 \\
 \frac{\Delta; \Gamma \vdash^e E_1 : B \rightarrow A \quad \Delta; \Gamma \vdash^e E_2 : B}{\Delta; \Gamma \vdash^e E_1 E_2 : A} \rightarrow E
 \end{array}$$

Modal Fragment

$$\begin{array}{c}
 \frac{u:A \text{ in } \Delta}{\Delta; \Gamma \vdash^e u : A} \text{mvar} \\
 \\
 \frac{\Delta; \cdot \vdash^e E : A}{\Delta; \Gamma \vdash^e \mathbf{box} E : \Box A} \Box I \\
 \\
 \frac{\Delta; \Gamma \vdash^e E_1 : \Box A \quad (\Delta, u:A); \Gamma \vdash^e E_2 : B}{\Delta; \Gamma \vdash^e \mathbf{let} \mathbf{box} u = E_1 \mathbf{in} E_2 : B} \Box E
 \end{array}$$

2.4. REDUCTION AND EXPANSION. The notions of β -reduction and η -expansion are fundamental to the λ -calculus. The preservation of types under β -reduction is the functional analog of local soundness for rules of natural deduction; the preservation of types under η -expansion is the functional analog of local completeness. But first we need to verify the characteristic properties of hypothetical judgments: exchange, weakening, contraction, and substitution.

LEMMA 1 (STRUCTURAL PROPERTIES OF CONTEXTS)

- (1) If $(\Delta_1, u:A, v:B, \Delta_2); \Gamma \vdash^e E:C$, then $(\Delta_1, v:B, u:A, \Delta_2); \Gamma \vdash^e E:C$.
- (2) If $\Delta; (\Gamma_1, x:A, y:B, \Gamma_2) \vdash^e E:C$, then $\Delta; (\Gamma_1, y:B, x:A, \Gamma_2) \vdash^e E:C$.
- (3) If $\Delta; \Gamma \vdash^e E:C$, then $(\Delta, u:A); \Gamma \vdash^e E:C$.
- (4) If $\Delta; \Gamma \vdash^e E:C$, then $\Delta; (\Gamma, x:A) \vdash^e E:C$.
- (5) If $(\Delta, u:A, v:A); \Gamma \vdash^e E:C$, then $(\Delta, w:A); \Gamma \vdash^e [w/u][w/v]E:C$.
- (6) If $\Delta; (\Gamma, x:A, y:A) \vdash^e E:C$, then $\Delta; (\Gamma, z:A) \vdash^e [z/x][z/y]E:C$.

PROOF. By straightforward inductions over the structure of the given derivations. Recall the global assumption that each variable is declared at most once in a context, and that bound variables may be renamed tacitly. \square

LEMMA 2 (SUBSTITUTION)

- (1) If $\Delta; \Gamma \vdash^e E_1:A$ and $\Delta; (\Gamma, x:A, \Gamma') \vdash^e E_2:B$
then $\Delta; (\Gamma, \Gamma') \vdash^e [E_1/x]E_2:B$.
- (2) If $\Delta; \cdot \vdash^e E_1:A$ and $(\Delta, u:A, \Delta'); \Gamma \vdash^e E_2:B$
then $(\Delta, \Delta'); \Gamma \vdash^e [E_1/u]E_2:B$.

PROOF. By straightforward inductions on the typing derivations for E_2 . \square

The β -reductions and η -expansions on proof terms used in the preceding section to verify local soundness and completeness are summarized below.

$$\begin{aligned}
 (\lambda x:A. E_2) E_1 &\xrightarrow{\beta} [E_1/x]E_2 \\
 \text{let box } u = \text{box } E_1 \text{ in } E_2 &\xrightarrow{\beta} [E_1/u]E_2 \\
 E:A \rightarrow B &\xrightarrow{\eta} \lambda x:A. Ex \\
 E:\Box A &\xrightarrow{\eta} \text{let box } u = E \text{ in box } u
 \end{aligned}$$

We now validate these rules by showing that they satisfy subject reduction.

THEOREM 3 (SUBJECT REDUCTION AND EXPANSION).

- (1) If $\Delta; \Gamma \vdash^e E:A$ and $E \xrightarrow{\beta} E'$, then $\Delta; \Gamma \vdash^e E':A$.
- (2) If $\Delta; \Gamma \vdash^e E:A$ and $E:A \xrightarrow{\eta} E'$, then $\Delta; \Gamma \vdash^e E':A$.

PROOF. In the case of a reduction, we first apply inversion (i.e., consider the possible final inferences in the derivation) and then use the substitution properties to obtain the result. In the case of an expansion, we directly construct the typing derivation for the expanded term. \square

We will not discuss commuting conversions for the $\Box E$ rule here, since they are not particularly relevant to our intended application. Similarly, we will not present a formal proof of strong normalization, although this is easy to obtain by an embedding into the ordinary simply-typed λ -calculus.

2.5. STAGED COMPUTATION. We now show the relationship between $\lambda_e^{\rightarrow \Box}$ and staged computation. It is our intention that those parts of a term enclosed by a

box constructor should be considered “uninterpreted code.” Thus, when we construct a computational interpretation of $\lambda_e^{\rightarrow\Box}$ based on β -reduction, it is appropriate to omit the congruence rule for **box**. We have the judgment:

$$E \mapsto E' \quad E \text{ reduces to } E'$$

This judgment is defined by the following rules.

$$\begin{array}{c} \frac{}{(\lambda x:A. E_2)E_1 \mapsto [E_1/x]E_2} \rightarrow \beta \\[10pt] \frac{}{\text{let } \mathbf{box} \ u = \mathbf{box} \ E_1 \text{ in } E_2 \mapsto [E_1/u]E_2} \Box\beta \\[10pt] \frac{E \mapsto E'}{\lambda x:A. E \mapsto \lambda x:A. E'} \text{cong_lam} \\[10pt] \frac{E_1 \mapsto E'_1}{E_1 E_2 \mapsto E'_1 E_2} \text{cong_app1} \\[10pt] \frac{E_2 \mapsto E'_2}{E_1 E_2 \mapsto E_1 E'_2} \text{cong_app2} \\[10pt] \frac{E_1 \mapsto E'_1}{\text{let } \mathbf{box} \ u = E_1 \text{ in } E_2 \mapsto \text{let } \mathbf{box} \ u = E'_1 \text{ in } E_2} \text{cong_letbox1} \\[10pt] \frac{E_2 \mapsto E'_2}{\text{let } \mathbf{box} \ u = E_1 \text{ in } E_2 \mapsto \text{let } \mathbf{box} \ u = E_1 \text{ in } E'_2} \text{cong_letbox2} \end{array}$$

We write \mapsto^* for the reflexive and transitive closure of \mapsto .

THEOREM 4 (SUBJECT REDUCTION WITH CONGRUENCES). *If $\Delta; \Gamma \vdash^e E:A$ and $E \mapsto^* E'$, then $\Delta; \Gamma \vdash^e E':A$.*

PROOF. By a simple induction on the derivation of $E \mapsto^* E'$, using subject reduction (Theorem 3) for the base cases. \square

To demonstrate the relationship between this reduction relation and computation staging, we roughly follow the binding-time correctness criteria described by Palsberg [1993]. Palsberg presented a modular proof of correctness for binding-time analyses based on two-level languages, such as those studied in Gomard and Jones 1991]. The first criterion is *consistency*, namely that static reduction of a well-annotated term cannot “go wrong.” In our case, well-annotated means well-typed, and the above subject reduction theorem corresponds roughly to the property of not “going wrong.” To make the correspondence more evident, we can simply note that a well-typed term cannot contain the “wrong” forms $(\mathbf{box} \ E'_1) \ E'_2$ and $\text{let } \mathbf{box} \ u = (\lambda x:A. E'_1) \text{ in } E'_2$.

The second criterion for binding-time correctness is that when a stage is complete, no subterm occurrences that are marked as eliminable remain. In our case, the subterm occurrences in the scope of a **box** constructor are code to be executed at a later stage and are therefore considered *persistent*; all other term occurrences are considered *eliminable*. Completing a stage means reducing a term until it can not be further reduced by the rules of the judgment $E \mapsto E'$. We call such terms *irreducible* to avoid confusion with subtly different notions

such as head-normal form or weak head-normal form. Note that an irreducible term could still contain a “redex” in the traditional sense underneath a **box** constructor. Since we only evaluate closed terms, the following theorem expresses that our language satisfies the second criterion for binding-time correctness.

THEOREM 5 (ELIMINABILITY). *If $\cdot; \cdot \vdash^e E : \Box A$ and $E \mapsto^* E'$ and E' is irreducible, then E' contains no eliminable term occurrences.*

PROOF. By subject reduction, $\cdot; \cdot \vdash^e E' : \Box A$. By inversion, and the fact that E' is irreducible, E' must have the form **box** E'_0 for some E'_0 . Therefore, all subterm occurrences in E' are in the scope of a **box** constructor and hence persistent. \square

Thus, it appears that Palsberg’s two properties both follow relatively easily from subject reduction for $\lambda_e^{\rightarrow\Box}$. However, there is still more to consider, because it is possible that eliminable subterms could reduce to persistent terms. This is ruled out syntactically in the two-level language studied by Palsberg, but in our case we need to show this explicitly. To argue about “images under reduction,” we temporarily extend $\lambda_e^{\rightarrow\Box}$ with labels.

$$\text{Terms } E ::= \dots | E^l$$

Labels have no impact on typing and can be reduced away.

$$\frac{\Delta; \Gamma \vdash^e E : A}{\Delta; \Gamma \vdash^e E^l : A} \text{LB}$$

$$\frac{}{E^l \mapsto E} \text{unlabel}$$

Recall that there is no congruence rule for **box** so that the rule **unlabel** can not be applied to a label in the scope of a **box** constructor.

Now suppose E_1 and E_2 differ only in that some subterms of E_1 have been labelled in E_2 . Then typing and reduction correspond between E_1 and E_2 . That is, $\Delta; \Gamma \vdash^e E_1 : A$ iff $\Delta; \Gamma \vdash^e E_2 : A$, and $E_1 \mapsto^* E'_1$ iff $E_2 \mapsto^* E'_2$ where E'_1 and E'_2 differ only in their labelling. This allows us to trace the “images under reduction” of eliminable parts of an unlabelled term by labelling all persistent subterm occurrences. The following theorem then expresses that only persistent parts of a term can yield persistent images. Here, for every subterm occurrence of the form E^l , both E and E^l are considered to be *labelled with l* . No other subterm occurrences are considered labelled.

THEOREM 6 (PERSISTENCE). *If $\Delta; \Gamma \vdash^e E : A$, all persistent subterm occurrences of E are labelled with l , and $E \mapsto E'$, then all persistent subterm occurrences of E' are labelled with l .*

PROOF. By induction on the derivation of $E \mapsto E'$.

Case:

$$\frac{}{(\lambda x:A. E_2)E_1 \mapsto [E_1/x]E_2} \rightarrow \beta$$

The result follows because the modal restriction in the typing rules does not allow x to appear in E_2 in a position enclosed by a **box** constructor. Formally, this case requires an auxiliary induction on E_2 .

Case:

$$\frac{}{\text{let box } u = \text{box } E_1 \text{ in } E_2 \mapsto [E_1/u]E_2} \Box\beta$$

The result follows because every subterm in E_1 is labelled with l .

Case:

$$\frac{E_2 \mapsto E'_2}{\text{let box } u = E_1 \text{ in } E_2 \mapsto \text{let box } u = E_1 \text{ in } E'_2} \text{cong_letbox2}$$

The result follows immediately by the induction hypothesis.

The other congruence cases are similar. If there were a congruence for **box**, that case would fail.

Case:

$$\frac{}{E^l \mapsto E} \text{unlabel}$$

The result is immediate, since E is not enclosed by **box** and therefore not persistent. \square

Interpreted as a statement about code manipulation during evaluation, this theorem says that we can never construct code from terms which were not originally code. This is one of the essential properties of $\lambda_e^{\rightarrow\Box}$ which makes it a suitable basis for languages allowing explicit code manipulation.

There is a dual property to persistence which is also enforced syntactically in the languages studied by Palsberg, namely that the eliminable parts of terms in the result of reduction only appear as the images of the eliminable parts of the original term. It is an important aspect of $\lambda_e^{\rightarrow\Box}$ is that it *does not* have this property, as shown by the counterexample:

$$\text{let box } u = \text{box } E \text{ in } u \mapsto E.$$

E appears in a persistent (or code) position on the left, but in an eliminable (or value) position on the right. From the point of view of code manipulation, it is easy to explain why this is allowed. In the languages we are interested in, the code representation for E can be *evaluated* to return a value for E , which stands in contrast to the languages studied by Palsberg.

Evaluation of code is expressed in $\lambda_e^{\rightarrow\Box}$ by occurrences of code variables u that are not enclosed by a **box** constructor. From a logical point of view, such instances correspond exactly to proofs which depend on the reflexivity of the Kripke reachability relation for modal S4. If we modify the rules to disallow such instances, we obtain the modal logic $K4$ and a corresponding modal λ -calculus which is somewhat closer to the two-level languages studied by Palsberg. If we also remove the transitivity of reachability from $K4$, we obtain the modal logic K and a corresponding modal λ -calculus which removes another feature of $\lambda_e^{\rightarrow\Box}$ that is not present in two-level languages, namely the ability to substitute code directly into code which is itself part of a code expression. So the following function from $\lambda_e^{\rightarrow\Box}$ would no longer be well-typed.

$$(\lambda x : \Box A. \text{let } \mathbf{box } u = \mathbf{box } E \text{ in } \mathbf{box } \mathbf{box } u) : \Box A \rightarrow \Box \Box A.$$

Allowing this feature in $\lambda_e^{\rightarrow\Box}$ seems reasonable and useful, though perhaps not as important as the inclusion of evaluation of code. We will come back to languages based on modal K later in Section 6, where we will briefly explain an exact correspondence between such languages and multi-level generalizations of two-level languages.

3. Modal Mini-ML: Explicit Formulation

This section presents Mini-ML_e[□], a language that combines some elements of Mini-ML [Clément et al. 1986] with the $\lambda_e^{\rightarrow\Box}$ -calculus of the previous section. For the sake of simplicity Mini-ML_e[□] is explicitly typed. ML-style or explicit polymorphism can also be added in a straightforward manner; we omit the details here in order to concentrate on the essential issues.

We present an operational semantics for the language, and demonstrate some basic properties such as type preservation. We also demonstrate the strong staging properties of the language. In the description of the operational semantics, we choose the usual device of representing values (including code) by corresponding source expressions. This may be refined in different ways for lower-level semantics describing, for example, run-time code generation or partial evaluation.

3.1. SYNTAX

Types	$A ::= \mathbf{nat} \mid A_1 \rightarrow A_2 \mid A_1 \times A_2 \mid 1 \mid \Box A$
Terms	$E ::= x \mid \lambda x:A. E \mid E_1 E_2$ $\quad \mid u \mid \mathbf{box } E \mid \text{let } \mathbf{box } u = E_1 \text{ in } E_2$ $\quad \mid \langle E_1, E_2 \rangle \mid \mathbf{fst } E \mid \mathbf{snd } E$ $\quad \mid \langle \rangle$ $\quad \mid \mathbf{z} \mid \mathbf{s } E \mid (\mathbf{case } E_1 \text{ of } \mathbf{z} \Rightarrow E_2 \mid \mathbf{s } x \Rightarrow E_3)$ $\quad \mid \mathbf{fix } x:A. E$
Ordinary Contexts	$\Gamma ::= \cdot \mid \Gamma, x:A$
Modal Contexts	$\Delta ::= \cdot \mid \Delta, u:A$

This language extends the one in the previous section, and we continue to use the conventions introduced in that section.

3.2. TYPING RULES. Our typing rules for the Mini-ML fragment of the explicit language are completely standard, and we follow the previous section for the modal fragment.

$\Delta; \Gamma \vdash^e E : A$ E has type A in modal context Δ and ordinary context Γ .

Functions

$$\begin{array}{c}
\frac{x:A \text{ in } \Gamma}{\Delta; \Gamma \vdash x : A} \text{tpe_ovar} \qquad \frac{\Delta; (\Gamma, x:A) \vdash E : B}{\Delta; \Gamma \vdash \lambda x:A. E : A \rightarrow B} \text{tpe_lam} \\
\\
\frac{\Delta; \Gamma \vdash E_1 : B \rightarrow A \quad \Delta; \Gamma \vdash E_2 : B}{\Delta; \Gamma \vdash E_1 E_2 : A} \text{tpe_app}
\end{array}$$

Code

$$\begin{array}{c}
\frac{u:A \text{ in } \Delta}{\Delta; \Gamma \vdash u : A} \text{tpe_mvar} \qquad \frac{\Delta; \cdot \vdash E : A}{\Delta; \Gamma \vdash \mathbf{box} E : \Box A} \text{tpe_box} \\
\\
\frac{\Delta; \Gamma \vdash E_1 : \Box A \quad (\Delta, u:A); \Gamma \vdash E_2 : B}{\Delta; \Gamma \vdash \mathbf{let box} u = E_1 \text{ in } E_2 : B} \text{tpe_let_box}
\end{array}$$

Products

$$\begin{array}{c}
\frac{\Delta; \Gamma \vdash E_1 : A_1 \quad \Delta; \Gamma \vdash E_2 : A_2}{\Delta; \Gamma \vdash \langle E_1, E_2 \rangle : A_1 \times A_2} \text{tpe_pair} \\
\\
\frac{\Delta; \Gamma \vdash E : A_1 \times A_2}{\Delta; \Gamma \vdash \mathbf{fst} E : A_1} \text{tpe_fst} \qquad \frac{\Delta; \Gamma \vdash E : A_1 \times A_2}{\Delta; \Gamma \vdash \mathbf{snd} E : A_2} \text{tpe_snd} \\
\\
\frac{}{\Delta; \Gamma \vdash \langle \rangle : 1} \text{tpe_unit}
\end{array}$$

Natural Numbers

$$\begin{array}{c}
\frac{}{\Delta; \Gamma \vdash \mathbf{z} : \mathbf{nat}} \text{tpe_z} \qquad \frac{\Delta; \Gamma \vdash E : \mathbf{nat}}{\Delta; \Gamma \vdash \mathbf{s} E : \mathbf{nat}} \text{tpe_s} \\
\\
\frac{\Delta; \Gamma \vdash E_1 : \mathbf{nat} \quad \Delta; \Gamma \vdash E_2 : A \quad \Delta; (\Gamma, x:\mathbf{nat}) \vdash E_3 : A}{\Delta; \Gamma \vdash (\mathbf{case} E_1 \text{ of } \mathbf{z} \Rightarrow E_2 \mid \mathbf{s} x \Rightarrow E_3) : A} \text{tpe_case}
\end{array}$$

Recursion

$$\frac{\Delta; \Gamma, x:A \vdash E : A}{\Delta; \Gamma \vdash \mathbf{fix} x:A. E : A} \text{tpe_fix}$$

As before, there is only one rule which introduces variables into the modal context (here called `tpe_let_box`).

3.3. OPERATIONAL SEMANTICS. The Mini-ML fragment of our system has a standard call-by-value operational semantics. For the modal part, we represent code for E simply by $\mathbf{box} E$, making the least commitment concerning lower-level implementations.

$$\text{Values } V ::= \lambda x:A. E \mid \langle V_1, V_2 \rangle \mid \langle \rangle \mid \mathbf{z} \mid \mathbf{s}V \mid \mathbf{box} E.$$

We evaluate $\mathbf{let box} x = E_1 \text{ in } E_2$ by substituting the code generated by evaluating E_1 for x in E_2 and then evaluating E_2 . The code generated by E_1 may then be evaluated during the evaluation of E_2 as necessary. On the λ -calculus and modal fragments, our semantics corresponds to a reduction strategy for $\lambda_e^{\rightarrow \Box}$.

$E \hookrightarrow V$ Expression E evaluates to value V .

Functions

$$\frac{}{\lambda x:A. E \hookrightarrow \lambda x:A. E} \text{ev_lam}$$

$$\frac{E_1 \hookrightarrow \lambda x. E'_1 \quad E_2 \hookrightarrow V_2 \quad [V_2/x]E'_1 \hookrightarrow V}{E_1 E_2 \hookrightarrow V} \text{ev_app}$$

Code

$$\frac{}{\mathbf{box} E \hookrightarrow \mathbf{box} E} \text{ev_box}$$

$$\frac{E_1 \hookrightarrow \mathbf{box} E'_1 \quad [E'_1/u]E_2 \hookrightarrow V_2}{\mathbf{let} \mathbf{box} u = E_1 \mathbf{in} E_2 \hookrightarrow V_2} \text{ev_let_box}$$

Products

$$\frac{E_1 \hookrightarrow V_1 \quad E_2 \hookrightarrow V_2}{\langle E_1, E_2 \rangle \hookrightarrow \langle V_1, V_2 \rangle} \text{ev_pair}$$

$$\frac{E \hookrightarrow \langle V_1, V_2 \rangle}{\mathbf{fst} E \hookrightarrow V_1} \text{ev_fst} \quad \frac{E \hookrightarrow \langle V_1, V_2 \rangle}{\mathbf{snd} E \hookrightarrow V_2} \text{ev_snd}$$

$$\frac{}{\langle \rangle \hookrightarrow \langle \rangle} \text{ev_unit}$$

Natural Numbers

$$\frac{}{\mathbf{z} \hookrightarrow \mathbf{z}} \text{ev_z} \quad \frac{E \hookrightarrow V}{\mathbf{s} E \hookrightarrow \mathbf{s} V} \text{ev_s}$$

$$\frac{E_1 \hookrightarrow \mathbf{z} \quad E_2 \hookrightarrow V}{(\mathbf{case} E_1 \mathbf{of} \mathbf{z} \Rightarrow E_2 \mid \mathbf{s} x \Rightarrow E_3) \hookrightarrow V} \text{ev_case_z}$$

$$\frac{E_1 \hookrightarrow \mathbf{s} V'_1 \quad [V'_1/x]E_3 \hookrightarrow V}{(\mathbf{case} E_1 \mathbf{of} \mathbf{z} \Rightarrow E_2 \mid \mathbf{s} x \Rightarrow E_3) \hookrightarrow V} \text{ev_case_s}$$

Recursion

$$\frac{[\mathbf{fix} x. E/x]E \hookrightarrow V}{\mathbf{fix} x. E \hookrightarrow V} \text{ev_fix}$$

The structural and substitution properties for $\lambda_e^{\rightarrow \square}$ extend to $\text{Mini-ML}_e^{\square}$ in a completely straightforward way, and we will make use of it below. We restate only the substitution lemma.

LEMMA 7 (SUBSTITUTION)

- (1) If $\Delta; \Gamma \vdash^e E_1:A$ and $\Delta; (\Gamma, x:A, \Gamma') \vdash^e E_2:B$
then $\Delta; (\Gamma, \Gamma') \vdash^e [E_1/x]E_2:B$.
- (2) If $\Delta; \cdot \vdash^e E_1:A$ and $(\Delta, u:A, \Delta'); \Gamma \vdash^e E_2:B$
then $(\Delta, \Delta'); \Gamma \vdash^e [E_1/u]E_2:B$.

PROOF. By straightforward inductions on the typing derivations for E_2 . \square

THEOREM 8 (DETERMINACY AND TYPE PRESERVATION).

- (1) If $E \hookrightarrow V$, then V is a value.
- (2) If $E \hookrightarrow V$ and $E \hookrightarrow \hat{V}$, then $V = \hat{V}$ (modulo renaming of bound variables).
- (3) If $E \hookrightarrow V$ and $\cdot; \cdot \vdash^e E : A$, then $\cdot; \cdot \vdash^e V : A$.

PROOF. By inductions over the structure of the derivation \mathcal{D} of $E \hookrightarrow V$. The cases for the nonmodal part are completely standard. The cases for `ev_box` are trivial and those for `ev_let_box` are straightforward for value soundness and determinacy. We thus show only the `ev_let_box` case in the proof of type preservation.

Case:

$$\mathcal{D} = \frac{\frac{\mathcal{D}_1}{E_1 \hookrightarrow \mathbf{box} E'_1} \quad \frac{\mathcal{D}_2}{[E'_1/u]E_2 \hookrightarrow V_2}}{\mathbf{let} \mathbf{box} u = E_1 \mathbf{in} E_2 \hookrightarrow V_2} \text{ev_let_box}$$

$\cdot; \cdot \vdash^e E_1 : \Box B$ and
 $u : B; \cdot \vdash^e E_2 : A$
 $\cdot; \cdot \vdash^e \mathbf{box} E'_1 : \Box B$
 $\cdot; \cdot \vdash^e E'_1 : B$
 $\cdot; \cdot \vdash^e [E'_1/u]E_2 : A$
 $\cdot; \cdot \vdash^e V_2 : A$

by inversion
 by ind. hyp.
 by inversion
 by substitution lemma
 by ind. hyp.

□

Since our semantics for Mini-ML_e^\Box is the natural extension of a reduction strategy for $\lambda_e^{\rightarrow\Box}$, the staging correctness results in Section 2 carry over to Mini-ML_e^\Box . We now briefly discuss the staging captured in Mini-ML_e^\Box in informal terms.

Suppose that $\cdot; \cdot \vdash^e E : \Box A$ and $E \hookrightarrow V$. By value soundness and type preservation we have $V \equiv \mathbf{box} E'$. Thus the result consists only of residual code to be executed in the next stage. Further, by the modal restrictions, only terms enclosed by `box` constructors are ever substituted into other `box` constructors. As a result, the parts of the original program E not enclosed by any `box` constructor can be designated eliminable (static) since they will not appear in the residual code E' .

Further, the body of a `box` constructor can be considered persistent (dynamic) in the sense that we do not evaluate underneath the `box` constructor. The only way for evaluation to proceed to the body of the `box` constructor is by using the variable bound by a `let box` elimination construct to indicate where the delayed computation should be performed.

3.4. EXAMPLES. The following functions demonstrate the interaction between products and \Box . They correspond to proofs in a modal logic with conjunction, since products and conjunction are related by the Curry–Howard isomorphism.

$\lambda x : (\Box A) \times (\Box B). \mathbf{let} \mathbf{box} u_1 = \mathbf{fst} x \mathbf{in} \mathbf{let} \mathbf{box} u_2 = \mathbf{snd} x \mathbf{in} \mathbf{box} \langle u_1, u_2 \rangle$
 $: (\Box A) \times (\Box B) \rightarrow \Box(A \times B)$

$\lambda x : \Box(A \times B). \mathbf{let} \mathbf{box} u = x \mathbf{in} \langle \mathbf{box} \mathbf{fst} u, \mathbf{box} \mathbf{snd} u \rangle$
 $: \Box(A \times B) \rightarrow (\Box A) \times (\Box B)$

We now define the power function in Mini-ML_e^\Box in such a way that it has type $\text{nat} \rightarrow \Box(\text{nat} \rightarrow \text{nat})$, assuming a closed term *times* : $\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ (definable

in the Mini-ML fragment in the standard way).

$$\begin{aligned} \text{power} &\equiv \text{fix } p:\text{nat} \rightarrow \Box(\text{nat} \rightarrow \text{nat}). \\ &\quad \lambda n:\text{nat}. \text{case } n \\ &\quad \text{of } \mathbf{z} \Rightarrow \text{box } (\lambda x:\text{nat}. \mathbf{s } \mathbf{z}) \\ &\quad \mid \mathbf{s } m \Rightarrow \text{let box } q = p \text{ m in} \\ &\quad \quad \text{box } (\lambda x:\text{nat}. \text{times } x (q \ x)) \end{aligned}$$

The type $\text{nat} \rightarrow \Box(\text{nat} \rightarrow \text{nat})$ expresses that *power* evaluates everything that depends on the first argument of type *nat* (the exponent) and returns residual code of type $\Box(\text{nat} \rightarrow \text{nat})$. Indeed, we calculate with our operational semantics:

$$\begin{aligned} \text{power } \mathbf{z} &\hookrightarrow \text{box } (\lambda x:\text{nat}. \mathbf{s } \mathbf{z}) \\ \text{power } (\mathbf{s } \mathbf{z}) &\hookrightarrow \text{box } (\lambda x:\text{nat}. \text{times } x ((\lambda x:\text{nat}. \mathbf{s } \mathbf{z})x)) \\ \text{power } (\mathbf{s } (\mathbf{s } \mathbf{z})) &\hookrightarrow \text{box } (\lambda x:\text{nat}. \text{times } x \\ &\quad ((\lambda x:\text{nat}. \text{times } x ((\lambda x:\text{nat}. \mathbf{s } \mathbf{z})x))x)) \end{aligned}$$

Modulo some trivial redices of variables for variables, this is the result we would expect from the partial evaluation of the *power* function.

3.5. IMPLEMENTATION ISSUES. The operational semantics of Mini-ML_e^\Box may be implemented by a translation into pure Mini-ML, by the mapping:

$$\begin{aligned} \Box A &\mapsto 1 \rightarrow A \\ \text{box } E &\mapsto \lambda x:1. E && (\text{where } x \text{ not free in } E) \\ \text{let box } u = E_1 \text{ in } E_2 &\mapsto (\lambda x:1 \rightarrow A. [x \langle \rangle / u] E_2) E_1 && (\text{where } x \text{ not in free } E_2) \end{aligned}$$

It may then appear that the modal fragment of Mini-ML_e^\Box is redundant. Note, however, that the type $1 \rightarrow A$ does not express any binding-time properties, while $\Box A$ does. It is precisely this distinction which makes Mini-ML_e^\Box interesting: The type checker will reject programs that may execute correctly, but for which the desired binding-time separation is violated. Without the modal operator, this property cannot be expressed and consequently not checked.

A more efficient implementation method would be to interpret $\Box A$ as a datatype representing code that calculates a value of type *A*. The representation must support substitution of one code fragment into another, as required by the *ev_let_box* rule. If the code is machine code, this naturally leads to the idea of templates, as used in run-time code generation (see Keppel et al. [1993]). For many applications this code would instead be source expressions or some intermediate language, thus allowing optimization after code substitution, as in partial evaluation [Jones et al. 1993]. In our own experiments in run-time code generation [Wickline et al. 1998a; 1998b] following ideas in Leone and Lee [1994] and Lee and Leone [1996], expressions of type $\Box A$ are compiled into *generating extensions* which emit machine code at run-time and then jump to it to effect evaluation. References to free code variables then represent calls from one generator to another.

4. A Kripke-Style Modal λ -Calculus

The modal logic in Section 2 was motivated by the goal to capture validity. A valid proposition is one with a closed proof term, and closed proof terms correspond to code which can be explicitly manipulated and safely evaluated.

In this section, we construct a modal logic based on Kripke's [1963] multiple-world interpretation of modal logic. A world corresponds to a stage of computa-

tion during evaluation. A value computed at a given stage of computation is available in all *accessible* stages, according to the accessibility relation between worlds of the Kripke semantics. Subtly different modal logics arise, depending on the properties of the accessibility relation between worlds. They are captured by structural rules built into the elimination rule for necessity ($\Box E$). In its most general form, we exactly capture validity and thereby the intuitionistic modal logic S4 presented in Section 2.

Our rules constitute a simplification of the system $\lambda^{\rightarrow\Box}$ in Pfenning and Wong [1995] and Davies and Pfenning [1996]. In particular, we have replaced the structural rule *pop* by a more general form of elimination which can be motivated from the perspective of pure natural deduction.

We prove the correctness of our system by relating it to the natural deduction system for S4 presented in Section 2 via two translations between proof terms. In Section 5, we extend this formulation of modal logic to Mini-ML $^\Box$, which leads to a staged programming style akin to Lisp's *quasiquote*, *unquote*, and *eval*. Instead of giving a direct operational semantics for this language we present a type preserving compilation to the explicit language from Section 3. We give an embedding of a two-level language [Nielson and Nielson 1992] into our language in Section 6.

4.1. A KRIPKE-STYLE NATURAL DEDUCTION SYSTEM. In Kripke's interpretation of modal logic, the truth of a proposition is relativized to a world. Modal operators allow us to reason about the truth of a proposition in all worlds *accessible* from the current world. Imposing laws on the accessibility relation between worlds (such as reflexivity, transitivity, or symmetry) leads to different modal logics. A world in the sense of Kripke is represented by a context of hypotheses containing propositions we know to be true in this world. Based on this intuition, our main judgment has the form

$$\Gamma_1; \Gamma_2; \dots; \Gamma_n \vdash^i A$$

which expresses that A is true in the current world Γ_n . Furthermore, Γ_1 represents hypotheses true in some initial world, Γ_2 represents hypotheses true in an *arbitrary* world reachable from Γ_1 , and so on.

From the functional point of view, Γ_i binds variables at stage i of the computation, where the proof term assigned to A may be executed at stage n . We refer to $\Gamma_1; \dots; \Gamma_n$ as the *world stack* or *context stack* since worlds are related to contexts by the Curry–Howard isomorphism. Note that there will always be at least one context in the context stack: the current world. We abbreviate (a possibly empty) context stack by Ψ .

As in Section 2, we now systematically develop a system of natural deduction for this judgment, which includes proof terms. First, only the hypotheses in the current world are available to derive the conclusion.

$$\frac{x:A \text{ in } \Gamma}{\Psi; \Gamma \vdash^i x:A} \text{var}$$

The substitution principle applies to arbitrary worlds, as long as we establish truth in the appropriate world.

Substitution Principle

If $\Psi; \Gamma \vdash^i M_1 : A$ and $\Psi; (\Gamma, x:A, \Gamma'); \Psi' \vdash^i M_2 : C$
 then $\Psi; (\Gamma, \Gamma'); \Psi' \vdash^i [M_1/x]M_2 : C$.

In the special case that Ψ' is empty, the current worlds in both given derivations coincide. We will formally demonstrate this property of the system later.

There are two kinds of structural properties. First, we have exchange, weakening and contraction within each world in the world stack. We will not formally restate this. Second, depending on the properties of the accessibility relation, we might have some structural properties on the world stack. In K we have none. If we add reflexivity of the accessibility relation, we reason as follows: If we have a judgment

$$\Psi; \Gamma; \Gamma'; \Psi' \vdash^i M : C,$$

then Γ' contains hypotheses assumed to be true in an *arbitrary* world accessible from Γ . But Γ itself is accessible from Γ (by reflexivity), so C should still be true if we join Γ and Γ' .

$$\Psi; (\Gamma, \Gamma'); \Psi' \vdash^i M' : C$$

Whether M' is different from M depends on how much information is present in the term itself, as we shall see later. We refer to this as *modal fusion*.

For example, omitting proof terms, we read the judgment

$$\Box(A \rightarrow B); A \vdash^i B$$

as

If $\Box(A \rightarrow B)$ is true in some world w_1 and A is true in an arbitrary world w_2 reachable from w_1 , then B is true in w_2 .

If we assume reflexivity of the accessibility relation, we know that w_1 is accessible from w_1 , so we can infer from the above by replacing w_2 with w_1 :

If $\Box(A \rightarrow B)$ is true in w_1 and A is true in w_1 , then B is true in w_1 .

In symbolic form, we write this as

$$\Box(A \rightarrow B), A \vdash^i B,$$

which is exactly the result of fusion applied to the first judgment.

Next we add transitivity. Consider once again

$$\Psi; \Gamma; \Gamma'; \Psi' \vdash^i M : C.$$

Then if we add an *arbitrary* world accessible from Γ from which Γ' can be reached, the judgment should continue to hold, because Γ' is still accessible from Γ by transitivity.

$$\Psi; \Gamma; \cdot; \Gamma'; \Psi' \vdash^i M' : C.$$

Again, M' may be identical to M or it can be directly created from M , depending how much information we represent in proof terms. We refer to this property as

modal weakening. Note that by ordinary weakening, the new interposed world may also contain arbitrary assumptions without invalidating the judgment.

Now we define the connectives via their introduction and elimination rules. Implication only affects the current world and is similar to what we have presented in Section 2.

$$\frac{\Psi; (\Gamma, x:A) \vdash^i M:B}{\Psi; \Gamma \vdash^i \lambda x:A. M:A \rightarrow B} \rightarrow I$$

$$\frac{\Psi; \Gamma \vdash^i M:A \rightarrow B \quad \Psi; \Gamma \vdash^i N:A}{\Psi; \Gamma \vdash^i MN:B} \rightarrow E$$

Local soundness and completeness also works as before. The corresponding operations on proof terms are the familiar β -reduction and η -expansion.

Recall that $\Box A$ should be true in the current world if A is true in every reachable world. Since we have no information about the reachable worlds, we have no hypotheses about the truth of propositions in this world. Hence, the introduction rule reads

$$\frac{\Psi; \Gamma; \cdot \vdash^i M:A}{\Psi; \Gamma \vdash^i \mathbf{box} M:\Box A} \Box I.$$

The corresponding elimination rule states that if $\Box A$ is true in the current world, A must be true in every reachable world. Which worlds are reachable depends, of course, on the accessibility relation for the modal logic under consideration. In its most general form (S4), the elimination rule reads

$$\frac{\Psi; \Gamma \vdash^i M:\Box A}{\Psi; \Gamma; \Gamma_1; \dots; \Gamma_n \vdash^i \mathbf{unbox}_n M:A} \Box E.$$

Note that Γ_1 is always accessible from Γ , so in K we only have \mathbf{unbox}_1 . The worlds $\Gamma_2, \dots, \Gamma_n$ are accessible from Γ only because of transitivity, so in modal logic with transitivity we also have \mathbf{unbox}_n for $n > 1$. Γ is accessible from itself in a modal logic whose accessibility relation is reflexive, so there we also have \mathbf{unbox}_0 .

Next we consider local soundness and completeness of the rules for the modal operator. Recall that soundness requires that an introduction rule followed by an elimination rule can be reduced to a direct derivation of the judgment.

$$\frac{\frac{\frac{\mathcal{D}}{\Psi; \Gamma; \cdot \vdash^i M:A}}{\Psi; \Gamma \vdash^i \mathbf{box} M:\Box A} \Box I}{\Psi; \Gamma; \Gamma_1; \dots; \Gamma_n \vdash^i \mathbf{unbox}_n (\mathbf{box} M):A} \Box E \quad \Rightarrow \quad \frac{\mathcal{D}'}{\Psi; \Gamma; \Gamma_1; \dots; \Gamma_n \vdash^i M':A}$$

where \mathcal{D}' and M' exist by the structural properties of world stacks (modal fusion in the case that $n = 0$, ordinary weakening in the case the $n = 1$, and ordinary weakening and $n - 1$ applications of modal weakening in the case that $n > 1$).

Local completeness is a bit simpler. We have the following η -expansion:

$$\Psi; \Gamma \vdash^{\mathcal{D}} M : \Box A \Rightarrow \frac{\frac{\Psi; \Gamma \vdash^{\mathcal{D}} M : \Box A}{\Psi; \Gamma; \cdot \vdash^{\Box E} \mathbf{unbox}_1 M : A} \Box E}{\Psi; \Gamma \vdash^{\Box I} \mathbf{box} (\mathbf{unbox}_1 M) : \Box A} \Box I$$

We postpone a formal discussion of the rules for β -reduction and η -expansion on terms to Section 4.4.

There are two simple and consistent variations on this system.

The first arises from an analysis of local completeness: one can see that only \mathbf{unbox}_1 is necessary. The others (which are locally sound!) have been incorporated so that we need no explicit structural rules. However, \mathbf{unbox}_1 plus explicit rules for modal fusion and weakening also make a sensible system with the same derivable judgments. For our purposes, a formulation without explicit structural rules is preferable, since it allows more compact programs and a simpler meta-theory.

In the second variant, we replace the constructor \mathbf{unbox}_n simply by \mathbf{unbox} . This would mean that $M' = M$ in the local reduction for \Box , and terms remain invariant under structural transformation of contexts. This more streamlined presentation of the calculus is not appropriate for our application, since the index n in a term $\mathbf{unbox}_n M$ determines the stage at which M is to be evaluated. Without the index, this would be ambiguous and depend on the typing derivation. In other words, the system would not be *coherent*.

4.2. SYNTAX. We summarize the syntax of the pure fragment.

Types	$A ::= a \mid A_1 \rightarrow A_2 \mid \Box A$
Terms	$M ::= x \mid \lambda x:A. M \mid M_1 M_2 \mid \mathbf{box} M \mid \mathbf{unbox}_n M$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x:A$
Context Stacks	$\Psi ::= \odot \mid \Psi; \Gamma$

Here and subsequently we write \odot for the empty context stack to avoid possible confusion with the empty context.

4.3. NATURAL DEDUCTION JUDGMENT. We summarize the rules defining the main judgment, $\Psi; \Gamma \vdash^i M:A$ as motivated and developed in Section 4.1.

$$\begin{array}{c} \frac{x:A \text{ in } \Gamma}{\Psi; \Gamma \vdash^i x : A} \text{var} \qquad \frac{\Psi; (\Gamma, x:A) \vdash^i M : B}{\Psi; \Gamma \vdash^i \lambda x:A. M : A \rightarrow B} \rightarrow I \\[10pt] \frac{\Psi; \Gamma \vdash^i M : A \rightarrow B \quad \Psi; \Gamma \vdash^i N : A}{\Psi; \Gamma \vdash^i M N : B} \rightarrow E \\[10pt] \frac{\Psi; \Gamma; \cdot \vdash^i M : A}{\Psi; \Gamma \vdash^i \mathbf{box} M : \Box A} \Box I \qquad \frac{\Psi; \Gamma \vdash^i M : \Box A}{\Psi; \Gamma; \Gamma_1; \dots; \Gamma_n \vdash^i \mathbf{unbox}_n M : A} \Box E \end{array}$$

4.4. PROPERTIES OF THE KRIPKE-STYLE Λ -CALCULUS. Structural transformations change the nature of the proof term by relabelling indices to the \mathbf{unbox} constructor. Such a relabelling is also necessary to write out the rules for

β -reduction and η -expansion. We define $\{n/m\}M$ inductively on the structure of M .

$$\begin{aligned}
 \{n/m\}x &= x \\
 \{n/m\}\lambda x:A. M &= \lambda x:A. \{n/m\}M \\
 \{n/m\}M_1M_2 &= (\{n/m\}M_1)(\{n/m\}M_2) \\
 \{n/m\}\mathbf{box}M &= \mathbf{box}\{n/m + 1\}M \\
 \{n/m\}\mathbf{unbox}_pM &= \begin{cases} \mathbf{unbox}_p\{n/m - p\}M & \text{for } p < m \\ \mathbf{unbox}_{p+n-1}M & \text{for } p \geq m \end{cases}
 \end{aligned}$$

This operation now allows us to state the substitution and structural transformation properties.

LEMMA 9 (MODAL STRUCTURAL TRANSFORMATION). *If $\Psi; \Gamma_0; \Delta_1; \dots; \Delta_m \vdash^i M:A$, then $\Psi; \Gamma_0; \dots; (\Gamma_n, \Delta_1); \dots; \Delta_m \vdash^i \{n/m\}M:A$.*

PROOF. By induction over the structure of the given derivation \mathcal{D} of

$$\Psi; \Gamma_0; \Delta_1; \dots; \Delta_m \vdash^i M:A.$$

In each case except for $\square E$, we immediately apply the induction hypothesis and reconstruct an appropriate derivation from the results. In the case of $\square E$, we distinguish two subcases.

Case:

$$\mathcal{D} = \frac{\Psi; \Gamma_0; \Delta_1; \dots; \Delta_{m-p} \vdash^i M_1 : \Box A}{\Psi; \Gamma_0; \Delta_1; \dots; \Delta_m \vdash^i \mathbf{unbox}_p M_1 : A} \square E$$

for $p < m$. Then

$$\begin{aligned}
 \Psi; \Gamma_0; \dots; (\Gamma_n, \Delta_1); \dots; \Delta_{m-p} \vdash^i \{n/m - p\}M_1 : \Box A & \quad \text{by ind. hyp.} \\
 \Psi; \Gamma_0; \dots; (\Gamma_n, \Delta_1); \dots; \Delta_m \vdash^i \mathbf{unbox}_p \{n/m - p\}M_1 : A & \quad \text{by rule } \square E \\
 \Psi; \Gamma_0; \dots; (\Gamma_n, \Delta_1); \dots; \Delta_m \vdash^i \{n/m\}\mathbf{unbox}_p M_1 : A & \quad \text{by definition of } \{n/m\}
 \end{aligned}$$

Case:

$$\mathcal{D} = \frac{\Psi'; \Theta_{p-m} \vdash^i M_1 : \Box A}{\Psi'; \Theta_{p-m}; \dots; \Theta_0; \Delta_1; \dots; \Delta_m \vdash^i \mathbf{unbox}_p M_1 : A} \square E$$

for $p \geq m$ where $\Psi = \Psi'; \dots; \Theta_1$ and $\Theta_0 = \Gamma_0$. Then

$$\begin{aligned}
 \Psi'; \Theta_{p-m}; \dots; \Theta_0; \Gamma_1; \dots; (\Gamma_n, \Delta_1); \dots; \Delta_m \vdash^i \mathbf{unbox}_{p+n-1} M_1 : A & \quad \text{by rule } \square E \text{ applied to } \mathcal{D}_1 \\
 \Psi; \Gamma_0; \dots; (\Gamma_n, \Delta_1); \dots; \Delta_m \vdash^i \{n/m\}\mathbf{unbox}_p M_1 : A & \quad \text{by definition of } \Psi, \Theta_0 \text{ and } \{n/m\}. \quad \square
 \end{aligned}$$

The system also satisfies the usual structural properties of exchange, weakening and contraction in each of the contexts in the context stack. We only state the substitution property formally.

LEMMA 10 (SUBSTITUTION). *If $\Psi; \Gamma \vdash^i M_1:A$ and $\Psi; (\Gamma, x:A, \Gamma'); \Psi' \vdash^i M_2:C$, then $\Psi; (\Gamma, \Gamma'); \Psi' \vdash^i [M_1/x]M_2:C$.*

PROOF. By induction over the structure of the derivation of $\Psi; (\Gamma, x:A, \Gamma'); \Psi' \vdash^i M_2:C$. \square

Then we have the rules of β -reduction and η -expansion, corresponding to local reduction and expansion in natural deduction.

$$\begin{array}{lcl}
 (\lambda x:A. M)N & \xrightarrow{\beta} & [N/x]M \\
 M:A \rightarrow B & \xrightarrow{\eta} & \lambda x:A. Mx \\
 \mathbf{unbox}_n(\mathbf{box} M) & \xrightarrow{\beta} & \{n/1\}M \\
 M:\Box A & \xrightarrow{\eta} & \mathbf{box}(\mathbf{unbox}_1 M).
 \end{array}$$

THEOREM 11 (SUBJECT REDUCTION AND EXPANSION)

- (1) If $\Psi; \Gamma \vdash^i M:A$ and $M \xrightarrow{\beta} M'$, then $\Psi; \Gamma \vdash^i M':A$.
- (2) If $\Psi; \Gamma \vdash^i M:A$ and $M:A \xrightarrow{\eta} M'$, then $\Psi; \Gamma \vdash^i M':A$

PROOF. In each case, we apply inversion to the given typing derivation. For subject reduction we then either use modal structural transformation (Lemma 9) or substitution (Lemma 10). For subject expansion we directly construct a derivation of the conclusion, using weakening if necessary. \square

4.5. ENVIRONMENTS AND ENVIRONMENT STACKS. In order to prove that the explicit (see Section 2) and implicit (Kripke style) formulations of S4 correspond, we need to develop some properties of environments and environment stacks. Roughly, an environment provides definitions for the modal variables available at a particular stage of the computation, while an environment stack extends this to all stages of a computation.

We define environments and stacks which bind patterns of the form $\mathbf{box} u$ to explicit terms E .

$$\begin{array}{ll}
 \text{Environments} & \rho ::= \cdot \mid \rho, \mathbf{box} u = E \\
 \text{Environment Stacks} & R ::= \odot \mid R; \rho
 \end{array}$$

The translations between the λ -calculi based on validity and multiple worlds require us to relate context pairs $\Delta; \Gamma$ to context stacks Ψ . This is achieved by the following typing judgments for environments and environment stacks. The latter ties in the context stacks of the implicit system. We use Θ to range over modal contexts.

$$\begin{array}{ll}
 \Delta; \Gamma \vdash^e \rho:\Theta & \text{Environment } \rho \text{ matches } \Theta \text{ in contexts } \Delta \text{ and } \Gamma \\
 \Psi \models^e R:\Delta & \text{Environment stack } R \text{ matches } \Delta \text{ in context stack } \Psi
 \end{array}$$

$$\begin{array}{c}
\frac{}{\Delta; \Gamma \vdash^e \cdot : \cdot} \text{tpv_empty} \\
\\
\frac{\Delta; \Gamma \vdash^e \rho : \Theta \quad (\Delta, \Theta); \Gamma \vdash^e E : \Box A}{\Delta; \Gamma \vdash^e (\rho, \mathbf{box} \ u = E) : (\Theta, u:A)} \text{tpv_bind} \\
\\
\frac{}{\Psi \models^e \odot : \cdot} \text{tpr_empty} \\
\\
\frac{\Psi \models^e R : \Delta \quad \Delta; \Gamma \vdash^e \rho : \Theta}{\Psi; \Gamma \models^e (R; \rho) : (\Delta, \Theta)} \text{tpr_env}
\end{array}$$

These rules are equivalent to the following noninductive ones, which are given here to aid the reader:

$$\begin{array}{c}
\frac{\forall 1 \leq i \leq n. \quad \Delta, u_1:A_1, \dots, u_{i-1}:A_{i-1}; \Gamma \vdash^e E_i : \Box A_i}{\Delta; \Gamma \vdash^e (\mathbf{box} \ u_1 = E_1, \dots, \mathbf{box} \ u_n = E_n) : (u_1:A_1, \dots, u_n:A_n)} \text{tpv_bind_n} \\
\\
\frac{\forall 1 \leq i \leq n. \quad (\Theta_1, \dots, \Theta_{i-1}); \Gamma_i \vdash^e \rho_i : \Theta_i}{\Psi; \Gamma_1; \dots; \Gamma_n \models^e (\odot; \rho_1; \dots; \rho_n) : (\Theta_1, \dots, \Theta_n)} \text{tpr_env_n}
\end{array}$$

We will tacitly use weakening for typing of environment stacks, which directly follows from weakening on the typing judgment for the explicit modal λ -calculus. We also need to use the following property.

LEMMA 12 (ENVIRONMENT EXTENSION). *If $\Psi; \Gamma \models^e (R; \rho) : \Delta$ and $\Delta; \Gamma \vdash^e E : \Box A$, then $\Psi; \Gamma \models^e (R; \rho, \mathbf{box} \ u = E) : (\Delta, u:A)$.*

PROOF. By inversion on the derivation for $(R; \rho)$ followed by a straightforward application of the typing rules for environments and environment stacks. \square

4.6. TRANSLATION FROM EXPLICIT SYSTEM. In this section, we show that if A is true in the explicit system then A is also true in the implicit system. We show this by giving a translation on proof terms. The difficulty in defining and proving the correctness of this translation lies in the relation between the modal and ordinary contexts on the explicit side and the context stack on the implicit side. This relationship can be maintained via the environment stacks defined in the preceding section.

$R; \rho \triangleright E \mapsto M$ Expression E translates to M in environment stack $R; \rho$

This judgment is defined by the following rules.

$$\begin{array}{c}
\frac{}{R; \rho \triangleright x \mapsto x} \text{tx_ovar} \quad \frac{R; \rho \triangleright E \mapsto M}{R; \rho \triangleright \lambda x:A. E \mapsto \lambda x:A. M} \text{tx_lam} \\
\\
\frac{R; \rho \triangleright E_1 \mapsto M_1 \quad R; \rho \triangleright E_2 \mapsto M_2}{R; \rho \triangleright E_1 E_2 \mapsto M_1 M_2} \text{tx_app} \\
\\
\frac{R; \rho; \cdot \triangleright E \mapsto M}{R; \rho \triangleright \mathbf{box} E \mapsto \mathbf{box} M} \text{tx_box} \\
\\
\frac{R; \rho, \mathbf{box} u = E_1 \triangleright E_2 \mapsto M}{R; \rho \triangleright \mathbf{let} \mathbf{box} u = E_1 \mathbf{in} E_2 \mapsto M} \text{tx_letbox} \\
\\
\frac{R; \rho'_n \triangleright E \mapsto M}{R; (\rho'_n, \mathbf{box} u = E, \rho''_n); \dots; \rho_0 \triangleright u \mapsto \mathbf{unbox}_n M} \text{tx_mvar}
\end{array}$$

THEOREM 13 (TRANSLATION FROM EXPLICIT SYSTEM). *Given $\Psi; \Gamma \models^e (R; \rho): \Delta$ and $\Delta; \Gamma \vdash^e E:A$.*

- (1) *There is a unique M such that $R; \rho \triangleright E \mapsto M$.*
- (2) *Whenever $R; \rho \triangleright E \mapsto M$, then $\Psi; \Gamma \vdash^i M:A$.*

PROOF. Proposition 1 is proven by induction on the multiset extension of the subterm ordering of expressions in $R; \rho$ and E . For the case of a modal variable u , we need to use the typing assumption to guarantee that tx_mvar applies, that is, that the environment stack contains an appropriate definition of u .

Proposition 2 is proven by induction on the structure of the derivation of $R; \rho \triangleright E \mapsto M$, applying inversion to the given typing derivations. In the case of modal variables u we have an auxiliary induction on the world index n .

We now show the proof of Proposition 2 in more detail. We assume we are given derivations

$$\begin{array}{ccccc}
\mathcal{V} & & \mathcal{E} & & \mathcal{T} \\
\Psi; \Gamma \models^e (R; \rho): \Delta & \text{and} & \Delta; \Gamma \vdash^e E:A & \text{and} & R; \rho \triangleright E \mapsto M
\end{array}$$

We proceed by induction on the structure of \mathcal{T} , applying inversion to the typing derivations as needed in order to construct a derivation

$$\begin{array}{c}
\mathcal{D} \\
\Psi; \Gamma \vdash^i M:A
\end{array}$$

Case:

$$\mathcal{T} = \frac{}{R; \rho \triangleright x \mapsto x} \text{tx_ovar}$$

$\Delta; \Gamma \models x : A$

$x:A$ in Γ

$\Psi; \Gamma \models x : A$

by assumption

by inversion

by rule var

Case:

$$\mathcal{T} = \frac{\frac{\mathcal{T}_2}{R; \rho \triangleright E_2 \mapsto M_2}}{R; \rho \triangleright \lambda x:A_1. E_2 \mapsto \lambda x:A_1. M_2} \text{tx_lam}$$

$\Delta; \Gamma \models \lambda x:A_1. E_2 : A$

$\Delta; \Gamma, x:A_1 \models E_2 : A_2$ and $A = A_1 \rightarrow A_2$

$\Psi; \Gamma, x:A_1 \models (R; \rho) : \Delta$

$\Psi; \Gamma, x:A_1 \models M_2 : A_2$

$\Psi; \Gamma \models \lambda x:A_1. M_2 : A_1 \rightarrow A_2$

by assumption

by inversion

by assumption and weakening

by ind. hyp.

by rule \rightarrow I

Case: tx_app is straightforward.

Case:

$$\mathcal{T} = \frac{\frac{\mathcal{T}_1}{R; \rho; \cdot \triangleright E_1 \mapsto M_1}}{R; \rho \triangleright \text{box } E_1 \mapsto \text{box } M_1} \text{tx_box}$$

$\Delta; \Gamma \models \text{box } E_1 : A$

$\Delta; \cdot \models E_1 : A_1$ and $A = \Box A_1$

$\Psi; \Gamma \models (R; \rho) : \Delta$

$\Delta; \cdot \models \cdot$

$\Psi; \Gamma; \cdot \models (R; \rho; \cdot) : \Delta$

$\Psi; \Gamma; \cdot \models M_1 : A_1$

$\Psi; \Gamma \models \text{box } M_1 : \Box A_1$

by assumption

by inversion

by assumption

by tpv_empty

by tpr_env

by ind. hyp.

by \Box I

Case:

$$\mathcal{T} = \frac{\frac{\mathcal{T}_1}{R; \rho, \text{box } u = E_1 \triangleright E_2 \mapsto M_2}}{R; \rho \triangleright \text{let box } u = E_1 \text{ in } E_2 \mapsto M_2} \text{tx_letbox}$$

$\Delta; \Gamma \models \text{let box } u = E_1 \text{ in } E_2 : A$

$\Delta; \Gamma \models E_1 : \Box A_1$ and

$(\Delta, u:A_1); \Gamma \models E_2 : A$

$\Psi; \Gamma \models (R; \rho) : \Delta$

$\Psi; \Gamma \models (R; \rho, \text{box } u = E_1) : (\Delta, u:A_1)$

$\Psi; \Gamma \models M_2 : A$

by assumption

by inversion

by assumption

by Lemma 12

by ind. hyp.

Case:

$$\mathcal{T} = \frac{\frac{\mathcal{T}_1}{R_n; \rho'_n \triangleright E' \mapsto M'}}{R_n; (\rho'_n, \text{box } u = E', \rho''_n); \dots; \rho_0 \triangleright u \mapsto \text{unbox}_n M'} \text{tx_mvar}$$

$\Delta; \Gamma \vdash^e u : A$	by assumption
$u : A$ in Δ	by inversion
$\Psi_0; \Gamma_0 \vdash^e R_n; (\rho'_n, \mathbf{box} u = E', \rho''_n); \dots; \rho_0 : \Delta$	
for $\Psi_0 = \Psi, \Gamma_0 = \Gamma$ and $\rho_0 = \rho$	by assumption
$\Delta = \Delta_0, \Theta_0,$	
$\Psi_0 \vdash^e R_n; (\rho'_n, \mathbf{box} u = E', \rho''_n); \dots; \rho_1 : \Delta_0,$ and	
$\Delta_0; \Gamma_0 \vdash^e \rho_0 : \Theta_0$	by inversion
$\Psi_{i-1} = \Psi_i; \Gamma_i,$	
$\Delta_{i-1} = \Delta_i, \Theta_i,$	
$\Psi_i \vdash^e R_n; (\rho'_n, \mathbf{box} u = E', \rho''_n); \dots; \rho_{i+1} : \Delta_i,$ and	
$\Delta_i; \Gamma_i \vdash^e \rho_i : \Theta_i$	by inversion for $i = 1, \dots, (n-1)$
$\Psi_{n-1} = \Psi_n; \Gamma_n,$	
$\Delta_{n-1} = \Delta_n, \Theta_n,$	
$\Psi_n \vdash^e R_n : \Delta_n,$ and	
$\Delta_n; \Gamma_n \vdash^e (\rho'_n, \mathbf{box} u = E', \rho''_n) : \Theta_n$	by further inversion
$\Theta_n = \Theta'_n, u : A', \Theta''_n,$	
$\Delta_n; \Gamma_n \vdash^e \rho'_n : \Theta'_n$ and	
$(\Delta_n, \Theta'_n); \Gamma_n \vdash^e E' : \Box A'$	by further inversion
$A' = A$	since $u : A$ in $\Delta = \Delta_n, \Theta_n, \dots, \Theta_0$ is unique
$\Psi_n; \Gamma_n \vdash^e (R_n; \rho'_n) : (\Delta_n, \Theta'_n)$	by rule <code>tpr_env</code>
$\Psi_n; \Gamma_n \vdash^i M' : \Box A$	by ind. hyp.
$\Psi_n; \Gamma_n; \dots; \Gamma_0 \vdash^i \mathbf{unbox}_n M' : A$	by rule <code>□E</code>

□

A special case of this theorem is the translation of a whole, well typed term: if $\cdot; \cdot \vdash^e E : A$ then there exists a unique M such that $\odot; \cdot \triangleright E \mapsto M$, and further we have that $\odot; \cdot \vdash^i M : A$.

4.7. TRANSLATION TO EXPLICIT SYSTEM. To show that every proposition judged true in the implicit system is also true in the explicit system, we give another type-preserving translation on proof terms. This translation is the core of the compilation function we consider in Section 5.4. Again, the difficulty lies mainly in relating the context stack of the implicit system to the modal and ordinary contexts of the explicit systems.

The translation recursively extracts terms inside \mathbf{unbox}_n constructors and binds their translation to new variables, bound with a **let box** outside the n th enclosing **box** constructor. Variables thus bound occur exactly once.

We abstract over an environment by means of nested **let box** expressions.

$$\begin{aligned} \text{Let}(\cdot)(E) &= E \\ \text{Let}(\rho, \mathbf{box} u = E')(E) &= \text{Let}(\rho)(\mathbf{let} \mathbf{box} u = E' \mathbf{in} E). \end{aligned}$$

We require a few straightforward properties of environments, but we explicitly state only the derived typing rule for environment abstractions.

$$\frac{\Delta; \Gamma \vdash^e \rho : \Theta \quad (\Delta, \Theta); \Gamma \vdash^e E : B}{\Delta; \Gamma \vdash^e \text{Let}(\rho)(E) : B} \text{tpr_env}$$

The merge operation $R_1 | R_2$ on environment stacks appends corresponding environments. We assume that the domains of the environments in R_1 and R_2 are disjoint so that the resulting environment stack is valid.

$$\begin{aligned}
\odot | R_2 &= R_2 \\
R_1 | \odot &= R_1 \\
(R_1; \rho_1) | (R_2; \rho_2) &= (R_1 | R_2); (\rho_1, \rho_2).
\end{aligned}$$

The translation is defined by the judgment

$$M \mapsto R \triangleright E \quad \text{\textit{M} compiles to term } E \text{ under stack } R$$

It is defined by the following rules.

$$\begin{array}{c}
\frac{}{x \mapsto \odot \triangleright x} \text{tr_var} \quad \frac{M \mapsto R \triangleright E}{\lambda x:A. M \mapsto R \triangleright \lambda x:A. E} \text{tr_lam} \\
\\
\frac{M_1 \mapsto R_1 \triangleright E_1 \quad M_2 \mapsto R_2 \triangleright E_2}{M_1 M_2 \mapsto (R_1 | R_2) \triangleright E_1 E_2} \text{tr_app} \\
\\
\frac{M \mapsto \odot \triangleright E}{\mathbf{box} M \mapsto \odot \triangleright \mathbf{box} E} \text{tr_box0} \quad \frac{M \mapsto (R; \rho) \triangleright E}{\mathbf{box} M \mapsto R \triangleright \text{Let}(\rho)(\mathbf{box} E)} \text{tr_box1} \\
\\
\frac{M \mapsto R \triangleright E}{\mathbf{unbox}_0 M \mapsto R \triangleright \mathbf{let} \mathbf{box} u = E \mathbf{in} u} \text{tr_unbox0} \\
\\
\frac{M \mapsto R \triangleright E}{\mathbf{unbox}_{n+1} M \mapsto R; (\mathbf{box} u = E); \underbrace{\cdot; \dots; \cdot}_n \triangleright u} \text{tr_unbox1}
\end{array}$$

The `tr_app` rule is restricted to context stacks R_1 and R_2 with disjoint domains. This can always be achieved by renaming of variables in the derivations of the two premisses.

THEOREM 14 (TRANSLATION TO EXPLICIT SYSTEM)

- (1) For any M there exist unique R and E such that $M \mapsto R \triangleright E$.
- (2) If $M \mapsto R \triangleright E$ and $\Psi; \Gamma \vdash^i M:A$, then for some Δ we have $\Psi \models^e R:\Delta$ and $\Delta; \Gamma \vdash^e E:A$.

PROOF. Proposition 1 is straightforward, since the translation is defined structurally on M with unique results (modulo renaming of bound variables, of course), except in the case of $\mathbf{box} M$, where exactly one of `tr_box0` and `tr_box1` apply.

Proposition 2 follows by induction on the structure of the derivation \mathcal{T} of $M \mapsto R \triangleright E$. The proof requires a few simple lemmas such as weakening for \vdash^e and some immediate properties of $R_1 | R_2$ and $\text{Let}(\rho)(E)$ which we do not state here explicitly. We omit the cases for the nonmodal constructors, which are straightforward.

Case:

$$\mathcal{T} = \frac{\mathcal{T}_1 \quad M_1 \mapsto \odot \triangleright E_1}{\mathbf{box} \ M_1 \mapsto \odot \triangleright \mathbf{box} \ E_1} \text{tr_box0}$$

$\Psi; \Gamma \vdash \mathbf{box} \ M_1 : A$ by assumption
 $\Psi; \Gamma; \cdot \vdash M_1 : A_1$ and $A = \Box A_1$ by inversion
 $\Psi; \Gamma \models \odot : \Delta_1$ and
 $\Delta_1; \cdot \models E_1 : A_1$ for some Δ_1 by ind. hyp.
 $\Delta_1 = \cdot$ by inversion
 $\Psi \models \odot : \cdot$ by tpr_empty
 $\cdot; \Gamma \models \mathbf{box} \ E_1 : \Box A_1$ by \Box

The last two lines are the desired conclusions for $\Delta = \cdot$.

Case:

$$\mathcal{T} = \frac{\mathcal{T}_1 \quad M_1 \mapsto (R; \rho) \triangleright E_1}{\mathbf{box} \ M_1 \mapsto R \triangleright \text{Let}(\rho)(\mathbf{box} \ E_1)} \text{tr_box1}$$

$\Psi; \Gamma \vdash \mathbf{box} \ M_1 : A$ by assumption
 $\Psi; \Gamma; \cdot \vdash M_1 : A_1$ and $A = \Box A_1$ by inversion
 $\Psi; \Gamma \models (R; \rho) : \Delta_1$ and
 $\Delta_1; \cdot \models E_1 : A_1$ for some Δ_1 by ind. hyp.
 $\Psi \models R : \Delta'_1$ and
 $\Delta'_1; \Gamma \models \rho : \Theta$ and $\Delta_1 = \Delta'_1, \Theta$ by inversion
 $(\Delta'_1, \Theta); \Gamma \models \mathbf{box} \ E_1 : \Box A_1$ by rule \Box
 $\Delta'_1; \Gamma \models \text{Let}(\rho)(\mathbf{box} \ E_1) : \Box A_1$ by rule tpi_env

Now we have the desired conclusions with $\Delta = \Delta'_1$.

Case:

$$\mathcal{T} = \frac{\mathcal{T}_1 \quad M_1 \mapsto R \triangleright E_1}{\mathbf{unbox}_0 \ M_1 \mapsto R \triangleright \mathbf{let} \ \mathbf{box} \ u = E_1 \ \mathbf{in} \ u} \text{tr_unbox0}$$

$\Psi; \Gamma \vdash \mathbf{unbox}_0 \ M_1 : A$ by assumption
 $\Psi; \Gamma \vdash M_1 : \Box A$ by inversion
 $\Psi \models R : \Delta_1$ and
 $\Delta_1; \Gamma \models E_1 : \Box A$ for some Δ_1 by ind. hyp.
 $(\Delta_1, u:A); \Gamma \models u : A$ by rule tpe_mvar
 $\Delta_1; \Gamma \models \mathbf{let} \ \mathbf{box} \ u = E_1 \ \mathbf{in} \ u : A$ by rule \Box

Now we have the desired conclusions with $\Delta = \Delta_1$.

Case:

$$\mathcal{T} = \frac{\mathcal{T}_1 \quad M_1 \mapsto R \triangleright E_1}{\mathbf{unbox}_{n+1} \ M_1 \mapsto R; (\mathbf{box} \ u = E_1); \underbrace{\cdot; \dots; \cdot}_{n} \triangleright u} \text{tr_unbox1}$$

$\Psi; \Gamma \vdash \mathbf{unbox}_{n+1} \ M_1 : A$ by assumption
 $\Psi'; \Gamma' \vdash M_1 : \Box A$ and $\Psi = \Psi'; \Gamma'; \Gamma_1; \dots; \Gamma_n$ by inversion
 $\Psi' \models R : \Delta_1$ and
 $\Delta_1; \Gamma' \models E_1 : \Box A$ for some Δ_1 by ind. hyp.
 $\Delta_1; \Gamma' \models \cdot : \cdot$ by rule tpv_empty
 $\Delta_1; \Gamma' \models (\mathbf{box} \ u = E_1) : (u : A)$ by rule tpv_bind
 $\Psi'; \Gamma' \models (R; \mathbf{box} \ u = E_1) : (\Delta_1, u:A)$ by rule tpr_env
 $\Psi'; \Gamma'_1; \dots; \Gamma_n \models (R; \mathbf{box} \ u = E_1; \cdot; \dots; \cdot) : (\Delta_1, u:A)$ by n applications of tpr_env
 $(\Delta_1, u:A); \Gamma \models u : A$ by tpe_mvar

Now we have the desired conclusions for $\Delta = \Delta_1, u:A$. \square

A special case of this theorem is the translation of a whole, well-typed term: if $\odot; \cdot \vdash^i M:A$, then there exists a unique E such that $M \mapsto \odot \triangleright E$, and further we have that $\cdot; \cdot \vdash^e E:A$. Here, the emptiness of the context stack forces the environment stack to be empty.

5. Modal Mini-ML: Implicit Formulation

We now define Mini-ML[□], an “implicit” formulation of modal Mini-ML generalizing the λ -calculus core from the preceding section. The main advantage of this system over the explicit language is that altering the staging of a computation in a given program often only requires the insertion or deletion of modal constructors. In contrast, Mini-ML_e[□] requires that the structure of the program exactly mirror the staging, since the only way to refer to results from a previous stage is via code variables. Using **let** (a derived form in our fragment) to bind code variables we can still express staging more explicitly in Mini-ML[□] if we prefer; it is now a matter of style rather than a property enforced in the language.

Another motivation for Mini-ML[□] is that it can be directly related to the two-level λ -calculus (see Section 6) which would be much more difficult for Mini-ML_e[□] due to the different syntactic structuring required. Further, Mini-ML[□] is very similar to the quasi-quoting and eval mechanisms in LISP, which are relatively intuitive in practice. We believe that with some syntactic sugar along the lines of Scheme’s backquote and comma notation (as in the regular expression example in Section 7.3), Mini-ML[□] is a practical and theoretically well-founded basis for an extension of Standard ML. Indeed, experience with the two languages PML [Wickline et al. 1998a] and Meta-ML [Taha and Sheard 1997; Taha et al. 1998; Moggi et al. 1999] indicates that such languages are indeed practical.

It may be helpful to consider the modal fragment of the implicit language to be a statically typed analog to the quasiquote mechanism in Scheme. Then **box** corresponds to quasiquote (‘) and **unbox**₁ to unquote (,). **unbox**₀ corresponds to eval. More generally, **unbox**_n corresponds to a generalized unquote which splices a quoted expression into a context with n levels of quasi-quoting. Note however that this analogy can also sometimes be misleading, and the actual behavior of code is closer to the quotations of a “semantically rationalized dialect” of Lisp called 2-Lisp [Smith 1984].

The operational semantics of the new system is given in terms of a type-preserving compilation to Mini-ML_e[□] which is a straightforward extension of the translation in Section 4.7.

For some applications, such as emulating the two-level λ -calculus, weaker modal logics such as K are sufficient, as described in Section 6.4.

5.1. SYNTAX. We extend the logic to the core of a programming language as in Section 3.

```

Types  A ::= nat | A1 → A2 | A1 × A2 | 1 | □A
Terms  M ::= x | λx:A. M | M1 M2
          | box M | unboxn M
          | ⟨M1, M2⟩ | fst M | snd M
          | ⟨⟩
          | z | s M | (case M1 of z ⇒ M2 | s x ⇒ M3)
          | fix x:A. M

Contexts  Γ ::= · | Γ, x:A
Context Stacks Ψ ::= ⊙ | Ψ; Γ

```

5.2. TYPING RULES. In this section we present typing rules for Mini-ML[□] using context stacks. The typing judgment has the form:

$\Psi; \Gamma \vdash^i M : A$ Term M has type A in local context Γ under stack Ψ .

Intuitively, each element Γ' of the context stack Ψ corresponds to a computation stage. The variables declared in Γ' are the ones whose values will be available during the corresponding evaluation phase. When we encounter a term **box** M during typing, we enter a new evaluation stage, since M will be frozen during evaluation of the current stage. In this new phase, we are not allowed to refer to variables of the prior phases, since they may not be available when **box** M is unfrozen using **unbox** _{n} . Thus, variables may only be looked up in the current context Γ (rule `tpi_var`) which is initialized as empty when we enter the body of a **box** (rule `tpi_box`). However, *code* generated in the current or earlier stages may be used, which is represented by the rule `tpi_unbox`.

Functions

$$\frac{x:A \text{ in } \Gamma}{\Psi; \Gamma \vdash^i x : A} \text{tpi_var} \quad \frac{\Psi; (\Gamma, x:A) \vdash^i M : B}{\Psi; \Gamma \vdash^i \lambda x:A. M : A \rightarrow B} \text{tpi_lam}$$

$$\frac{\Psi; \Gamma \vdash^i M : A \rightarrow B \quad \Psi; \Gamma \vdash^i N : A}{\Psi; \Gamma \vdash^i M N : B} \text{tpi_app}$$

Code

$$\frac{\Psi; \Gamma; \cdot \vdash^i M : A}{\Psi; \Gamma \vdash^i \text{box } M : \Box A} \text{tpi_box} \quad \frac{\Psi; \Gamma \vdash^i M : \Box A}{\Psi; \Gamma; \Gamma_1; \dots; \Gamma_n \vdash^i \text{unbox}_n M : A} \text{tpi_unbox}$$

Products

$$\frac{\Psi; \Gamma \vdash^i M_1 : A_1 \quad \Psi; \Gamma \vdash^i M_2 : A_2}{\Psi; \Gamma \vdash^i \langle M_1, M_2 \rangle : A_1 \times A_2} \text{tpi_pair}$$

$$\frac{\Psi; \Gamma \vdash^i M : A_1 \times A_2}{\Psi; \Gamma \vdash^i \text{fst } M : A_1} \text{tpi_fst} \quad \frac{\Psi; \Gamma \vdash^i M : A_1 \times A_2}{\Psi; \Gamma \vdash^i \text{snd } M : A_2} \text{tpi_snd}$$

$$\frac{}{\Psi; \Gamma \vdash^i \langle \rangle : 1} \text{tpi_unit}$$

Natural Numbers

$$\frac{}{\Psi; \Gamma \vdash^i z : \text{nat}} \text{tpi_z} \quad \frac{\Psi; \Gamma \vdash^i M : \text{nat}}{\Psi; \Gamma \vdash^i sM : \text{nat}} \text{tpi_s}$$

$$\frac{\Psi; \Gamma \vdash^i M_1 : \text{nat} \quad \Psi; \Gamma \vdash^i M_2 : A \quad \Psi; (\Gamma, x:\text{nat}) \vdash^i M_3 : A}{\Psi; \Gamma \vdash^i (\text{case } M_1 \text{ of } z \Rightarrow M_2 \mid s x \Rightarrow M_3) : A} \text{tpi_case}$$

Recursion

$$\frac{\Psi; (\Gamma, x:A) \vdash^i M : A}{\Psi; \Gamma \vdash^i \text{fix } x:A. M : A} \text{tpi_fix}$$

The reductions and expansions from Section 4 remain valid in this extended setting, as do the structural rules.

5.3. EXAMPLES IN IMPLICIT FORM. We now show how we can define the power function in Mini-ML^\square with a different syntactic structure than in Mini-ML_e^\square , though still with type $\text{nat} \rightarrow \square(\text{nat} \rightarrow \text{nat})$.

$$\begin{aligned} \text{power} &\equiv \mathbf{fix} \, p : \text{nat} \rightarrow \square(\text{nat} \rightarrow \text{nat}). \\ &\quad \lambda n : \text{nat}. \mathbf{case} \, n \\ &\quad \quad \mathbf{of} \, z \Rightarrow \mathbf{box}(\lambda x : \text{nat}. \mathbf{sz}) \\ &\quad \quad | sm \Rightarrow \mathbf{box}(\lambda x : \text{nat}. \text{times } x(\mathbf{unbox}_1(pm)x)) \end{aligned}$$

As another example, we show how to define a function of type $\text{nat} \rightarrow \square \text{nat}$ that returns a **box**'ed copy of its argument:

$$\begin{aligned} \text{lift}_{\text{nat}} &\equiv \mathbf{fix} \, f : \text{nat} \rightarrow \square \text{nat}. \\ &\quad \lambda x : \text{nat}. \mathbf{case} \, x \\ &\quad \quad \mathbf{of} \, z \Rightarrow \mathbf{box} \, z \\ &\quad \quad | sx' \Rightarrow \mathbf{box}(s(\mathbf{unbox}_1(fx'))) \end{aligned}$$

A similar term of type $A \rightarrow \square A$ that returns a **box**'ed copy of its argument exists exactly when every \rightarrow in A is enclosed by a \square . This justifies the inclusion of the *lift* primitive for base types in two-level languages such as in Gomard and Jones [1991], and it seems natural to include such a primitive in a realistic extension of our language, as in Wickline et al. [1998a].

5.4. COMPILATION TO EXPLICIT LANGUAGE. We do not define an operational semantics for Mini-ML^\square directly; instead we depend upon a translation to Mini-ML_e^\square . This extends the translation given in Section 4.7 in a straightforward way. We prefer this to a direct operational semantics on the implicit language since the translation should be identical to what a compiler would perform. We omit the obvious rules.

As an example of the compilation, it maps the definition of *power* from Section 5.3 to the one in Section 3.4. Note that the restructuring achieved by the compiler is similar to a staging transformation [Jørring and Scherlis 1986].

The operational semantics induced by the translation is different from some obvious ones defined directly on Mini-ML^\square . In Martini and Masini [1996], for example, a simple reduction semantics is introduced for a system similar to the pure fragment of our implicit system. It does not reflect staging, and is instead used to prove a Church–Rosser theorem and strong normalization for a pure modal λ -calculus. Similarly, in Pfenning and Wong [1995], an algorithm for converting pure modal λ -terms in implicit form to long normal form is given and proven correct. This algorithm bears no resemblance to the staged computation achieved via Mini-ML_e^\square . We also have constructed a direct operational semantics for Mini-ML^\square generalizing [Hatcliff 1995] that does capture staging, but prefer the compilation because it makes operational properties more evident. In particular, proving staging theorems for Mini-ML^\square directly would be much harder than taking advantage of the type-preserving compilation and proving the properties for Mini-ML_e^\square .

6. A Two-Level Language

In this section, we define Mini-ML₂, a two-level functional language very close to the one described in Nielson and Nielson [1992]. We then define a simple translation into Mini-ML[□] and prove that binding-time correctness in Mini-ML₂ is equivalent to modal correctness of the translation in Mini-ML[□].

A two-level language captures staging by explicitly annotating each occurrence of a term constructor as *compile-time* (often called *static*) or *run-time* (often called *dynamic*). Traditionally, expression constructors that can be evaluated at compile-time are overlined, those which cannot be evaluated until run-time are underlined. The process of annotating each term constructor in an expression is called *binding-time analysis*. Of course, not every possible annotation is valid. For example, the expression

$$\underline{\lambda}x : \underline{\text{nat}}. \overline{\text{case } x \text{ of } \underline{z} \Rightarrow \underline{z} \mid \underline{sy} \Rightarrow \underline{sz}}$$

is not binding-time correct, since \underline{x} is not available until run-time, while the case statement is annotated to be executed at compile time, which is not possible.

We will not discuss binding-time analysis in this paper, only show how the resulting two-level terms are related to modal Mini-ML in its multiple-world formulation from Section 5.

Our language differs slightly from Nielson and Nielson [1992] in that we inject *all* run-time types into compile-time types, instead of just function types. This follows [Gomard and Jones 1991], where there is no such restriction. Also, we find it convenient to divide the variables and contexts into run-time and compile-time. All other differences to Nielson and Nielson [1992] are due to minor differences between their underlying language and Mini-ML. Note that modal Mini-ML can accommodate arbitrary levels (not just two) and additional term operations (such as evaluation), so the two-level language we introduce in this section will be embedded into a relatively modest fragment of modal Mini-ML.

6.1. SYNTAX

Run-time Types	$\tau ::= \underline{\text{nat}} \mid \tau_1 \Rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \underline{1}$
Compile-time Types	$\sigma ::= \overline{\text{nat}} \mid \sigma_1 \Rightarrow \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \overline{1} \mid \tau$
Terms	$ \begin{aligned} e ::= & \underline{x} \mid \underline{\lambda} \underline{x} : \tau. e \mid e_1 @ e_2 \\ & \mid \underline{\text{fix}} \underline{x} : \tau. e \\ & \mid \langle \underline{e}_1, \underline{e}_2 \rangle \mid \underline{\text{fst}} e \mid \underline{\text{snd}} e \\ & \mid \underline{z} \mid \underline{s} e \\ & \mid (\underline{\text{case}} e_1 \text{ of } \underline{z} \Rightarrow e_2 \mid \underline{s} \underline{x} \Rightarrow e_3) \\ & \mid \langle \rangle \\ & \mid \overline{y} \mid \overline{\lambda} \overline{y} : \sigma. e \mid e_1 @ e_2 \\ & \mid \overline{\text{fix}} \overline{y} : \sigma. e \\ & \mid \langle \overline{e}_1, \overline{e}_2 \rangle \mid \overline{\text{fst}} e \mid \overline{\text{snd}} e \\ & \mid \langle \rangle \\ & \mid \overline{z} \mid \overline{s} e \\ & \mid (\overline{\text{case}} e_1 \text{ of } \overline{z} \Rightarrow e_2 \mid \overline{s} \overline{y} \Rightarrow e_3) \end{aligned} $
Run-time Contexts	$\Gamma ::= \cdot \mid \Gamma, \underline{x} : \tau$
Compile-time Contexts	$\Delta ::= \cdot \mid \Delta, \overline{y} : \sigma$

As a simple example we consider the two-level version of the power function.

$$\begin{aligned}
 \text{power} \quad &\equiv \quad \overline{\text{fix}} \, \overline{p} : \overline{\text{nat}} \Rightarrow (\overline{\text{nat}} \Rightarrow \overline{\text{nat}}). \\
 &\quad \overline{\lambda n : \text{nat. case } n} \\
 &\quad \overline{\text{of } z \Rightarrow (\lambda x : \text{nat. } \underline{\text{sz}})} \\
 &\quad |\underline{\text{sm}} \Rightarrow (\lambda x : \text{nat. } \underline{\text{times}} \, @ \, x \, @ \, ((\overline{p} \, \overline{@} \, \overline{m}) @ \, x))
 \end{aligned}$$

Recall that *times* is a curried function for multiplication represented as a closed term for simplicity. The type indicates that *power* takes a natural number as a compile-time argument and computes a residual run-time function from *nat* to *nat*. Otherwise, the structure is very similar to the *power* function in its implicit formulation from Section 5.3. As we will see in Section 6.3 we can translate this to Mini-ML[□] by inserting a **box** constructor when an immediate subexpression of a compile-time term (overlined) is a run-time term (underlined). Conversely, when a compile-time term appears as an immediate subexpression of a run-time term we insert an **unbox**₁ constructor. It is easy to see that in this example we obtain the *power* function in implicit form, exactly as in Section 5.3:

$$\begin{aligned}
 \text{power} \quad &\equiv \quad \text{fix } p : \text{nat} \rightarrow \square(\text{nat} \rightarrow \text{nat}). \\
 &\quad \lambda n : \text{nat. case } n \\
 &\quad \text{of } z \Rightarrow \text{box}(\lambda x : \text{nat. } \underline{\text{sz}}) \\
 &\quad |\underline{\text{sm}} \Rightarrow \text{box}(\lambda x : \text{nat. } \underline{\text{times}} \, x (\underline{\text{unbox}}_1(p \, m) \, x)).
 \end{aligned}$$

6.2. TYPING RULES. The typing rules of the two-level λ -calculus simultaneously verify staging and standard type-correctness, just as our explicit and implicit systems. We have two judgments:

$$\begin{array}{ll}
 \Delta; \Gamma \vdash^r e : \tau & \text{expression } e \text{ has run-time type } \tau \\
 \Delta \vdash^c e : \sigma & \text{expression } e \text{ has compile-time type } \sigma
 \end{array}$$

A compile-time expression can never depend on a run-time variable. Therefore, compile-time typing depends only on a compile-time context. A run-time expression may have embedded compile-time subexpressions and therefore carries compile-time variables (in Δ) as well as run-time variables in Γ .

Functions

$$\begin{array}{c}
\frac{\underline{x}:\tau \text{ in } \Gamma}{\Delta; \Gamma \vdash \underline{x} : \tau} \text{ tpr_var} \qquad \frac{\Delta; (\Gamma, \underline{x}:\tau_2) \vdash e : \tau}{\Delta; \Gamma \vdash \lambda \underline{x}:\tau_2. e : \tau_2 \Rightarrow \tau} \text{ tpr_lam} \\
\\
\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \Rightarrow \tau \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1 @ e_2 : \tau} \text{ tpr_app}
\end{array}$$

Products

$$\begin{array}{c}
\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \text{ tpr_pair} \\
\\
\frac{\Delta; \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash \text{fst } e : \tau_1} \text{ tpr_fst} \qquad \frac{\Delta; \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash \text{snd } e : \tau_2} \text{ tpr_snd} \\
\\
\frac{}{\Delta; \Gamma \vdash \langle \rangle : \underline{1}} \text{ tpr_unit}
\end{array}$$

Natural Numbers

$$\begin{array}{c}
\frac{}{\Delta; \Gamma \vdash \underline{z} : \text{nat}} \text{ tpr_z} \qquad \frac{\Delta; \Gamma \vdash e : \text{nat}}{\Delta; \Gamma \vdash \underline{s } e : \text{nat}} \text{ tpr_s} \\
\\
\frac{\Delta; \Gamma \vdash e_1 : \text{nat} \quad \Delta; \Gamma \vdash e_2 : \tau \quad \Delta; (\Gamma, \underline{x} : \text{nat}) \vdash e_3 : \tau}{\Delta; \Gamma \vdash (\text{case } e_1 \text{ of } \underline{z} \Rightarrow e_2 \mid \underline{s } \underline{x} \Rightarrow e_3) : \tau} \text{ tpr_case}
\end{array}$$

Recursion

$$\frac{\Delta; (\Gamma, \underline{x}:\tau) \vdash e : \tau}{\Delta; \Gamma \vdash \text{fix } \underline{x}:\tau. e : \tau} \text{ tpr_fix}$$

Phase Transitions

$$\frac{\Delta \vdash^e e : \tau}{\Delta; \Gamma \vdash^e e : \tau} \text{ down} \qquad \frac{\Delta; \cdot \vdash^e e : \tau}{\Delta \vdash^e e : \tau} \text{ up}$$

Functions

$$\begin{array}{c}
\frac{\bar{y}:\sigma \text{ in } \Delta}{\Delta \vdash^e \bar{y} : \sigma} \text{ tpc_var} \qquad \frac{\Delta, \bar{y}:\sigma_2 \vdash^e e : \sigma}{\Delta \vdash^e \bar{\lambda} \bar{y}:\sigma_2. e : \sigma_2 \Rightarrow \sigma} \text{ tpc_lam} \\
\\
\frac{\Delta \vdash^e e_1 : \sigma_2 \Rightarrow \sigma \quad \Delta \vdash^e e_2 : \sigma_2}{\Delta \vdash^e e_1 @ e_2 : \sigma} \text{ tpc_app}
\end{array}$$

Products

$$\begin{array}{c}
\frac{\Delta \vdash^e e_1 : \sigma_1 \quad \Delta \vdash^e e_2 : \sigma_2}{\Delta \vdash^e \overline{\langle e_1, e_2 \rangle} : \sigma_1 \overline{\times} \sigma_2} \text{ tpc_pair} \\
\\
\frac{\Delta \vdash^e e : \sigma_1 \overline{\times} \sigma_2}{\Delta \vdash^e \overline{\text{fst}} e : \sigma_1} \text{ tpc_fst} \qquad \frac{\Delta \vdash^e e : \sigma_1 \overline{\times} \sigma_2}{\Delta \vdash^e \overline{\text{snd}} e : \sigma_2} \text{ tpc_snd} \\
\\
\frac{}{\Delta \vdash^e \overline{\langle \rangle} : \overline{1}} \text{ tpc_unit}
\end{array}$$

Natural Numbers

$$\begin{array}{c}
\frac{}{\Delta \vdash^c \bar{z} : \bar{\text{nat}}} \text{tpc.z} \qquad \frac{\Delta \vdash^c e : \bar{\text{nat}}}{\Delta \vdash^c \bar{s} e : \bar{\text{nat}}} \text{tpc.s} \\
\\
\frac{\Delta \vdash^c e_1 : \bar{\text{nat}} \quad \Delta \vdash^c e_2 : \sigma \quad \Delta, \bar{y} : \bar{\text{nat}} \vdash^c e_3 : \sigma}{\Delta \vdash^c (\text{case } e_1 \text{ of } \bar{z} \Rightarrow e_2 \mid \bar{s} \bar{y} \Rightarrow e_3) : \sigma} \text{tpc.case}
\end{array}$$

Recursion

$$\frac{\Delta, \bar{y} : \sigma \vdash^c e : \sigma}{\Delta \vdash^c \bar{\text{fix}} \bar{y} : \sigma. e : \sigma} \text{tpc.fix}$$

Note that we remove run-time assumptions at the down rule, while in Nielson and Nielson [1992] this is done later at the up rule, via a rule like:

$$\frac{\Delta; \cdot \vdash^r e : \tau}{\Delta; \Gamma \vdash^c e : \tau} \text{up.}$$

Our change is justified since, by the structure of their rules, such assumptions can never be used in the compile-time deduction in between.

6.3. TRANSLATION TO IMPLICIT LANGUAGE. The translation to Mini-ML[□] is now very simple. We translate both run-time and compile-time Mini-ML fragments directly, and insert \square , **box** and **unbox**₁ to represent the changes between phases. We define two mutually recursive functions to do this: $\|\cdot\|$ is the run-time translation and $|\cdot|$ is the compile-time translation. We overload this notation by using it for types, terms, and contexts. We write \underline{e} and \bar{e} to match any term whose top constructor matches the phase annotation.

Type Translation

$$\begin{array}{ll}
\|\mathbf{nat}\| = \mathbf{nat} & |\overline{\mathbf{nat}}| = \mathbf{nat} \\
\|\tau_1 \rightarrow \tau_2\| = \|\tau_1\| \rightarrow \|\tau_2\| & |\sigma_1 \rightarrow \sigma_2| = |\sigma_1| \rightarrow |\sigma_2| \\
\|\tau_1 \times \tau_2\| = \|\tau_1\| \times \|\tau_2\| & |\sigma_1 \times \sigma_2| = |\sigma_1| \times |\sigma_2| \\
\|\mathbf{1}\| = 1 & |\overline{\mathbf{1}}| = 1 \\
& |\tau| = \Box \|\tau\|
\end{array}$$

Term Translation

$$\begin{array}{ll}
\|\underline{x}\| = x & |\overline{y}| = y \\
\|\lambda \underline{x}:\tau. e\| = \lambda x:\|\tau\|. \|e\| & |\overline{\lambda \overline{y}:\sigma. e}| = \lambda y:|\sigma|. |e| \\
\|e_1 @ e_2\| = \|e_1\| \|e_2\| & |e_1 @ e_2| = |e_1| |e_2| \\
\|\mathbf{fix} \underline{x}:\tau. e\| = \mathbf{fix} x:\|\tau\|. \|e\| & |\overline{\mathbf{fix} \overline{y}:\sigma. e}| = \mathbf{fix} y:|\sigma|. |e| \\
\|\langle e_1, e_2 \rangle\| = \langle \|e_1\|, \|e_2\| \rangle & |\overline{\langle e_1, e_2 \rangle}| = \langle |e_1|, |e_2| \rangle \\
\|\mathbf{fst} e\| = \mathbf{fst} \|e\| & |\overline{\mathbf{fst} e}| = \mathbf{fst} |e| \\
\|\mathbf{snd} e\| = \mathbf{snd} \|e\| & |\overline{\mathbf{snd} e}| = \mathbf{snd} |e| \\
\|\langle \rangle\| = \langle \rangle & |\overline{\langle \rangle}| = \langle \rangle \\
\|\underline{z}\| = \mathbf{z} & |\overline{z}| = \mathbf{z} \\
\|\underline{s} e\| = \mathbf{s} \|e\| & |\overline{s e}| = \mathbf{s} |e|
\end{array}$$

$$\begin{array}{l}
\|\mathbf{case} e_1 \mathbf{of} \underline{z} \Rightarrow e_2 \mid \underline{s} \underline{x} \Rightarrow e_3\| = \\
\mathbf{case} \|e_1\| \mathbf{of} \mathbf{z} \Rightarrow \|e_2\| \mid \mathbf{s} x \Rightarrow \|e_3\| \\
|\overline{\mathbf{case} e_1 \mathbf{of} \overline{z} \Rightarrow e_2 \mid \overline{s} \overline{y} \Rightarrow e_3}| = \\
\mathbf{case} |e_1| \mathbf{of} \mathbf{z} \Rightarrow |e_2| \mid \mathbf{s} y \Rightarrow |e_3|
\end{array}$$

$$\|\overline{e}\| = \mathbf{unbox}_1 |\overline{e}| \qquad |e| = \mathbf{box} \|\overline{e}\|$$

Context Translation

$$\begin{array}{ll}
\|\cdot\| = \cdot & |\cdot| = \cdot \\
\|\Gamma, \underline{x}:\tau\| = \|\Gamma\|, x:\|\tau\| & |\Delta, \overline{y}:\sigma| = |\Delta|, y:|\sigma|
\end{array}$$

6.4. EQUIVALENCE OF BINDING TIME CORRECTNESS AND MODAL CORRECTNESS. In this section, we show that binding-time correctness is equivalent to modal correctness of the translation to Mini-ML[□]. Note that even though we use Δ and Γ to denote contexts, the implicit language Mini-ML[□] employs context stacks, where $\odot; \Gamma_1; \dots; \Gamma_n$ is abbreviated as $\Gamma_1; \dots; \Gamma_n$.

THEOREM 15 (CONSERVATIVE EMBEDDING)

(1) If $\|e\| = M$, then:

- (a) if $\Delta; \Gamma \vdash^r e:\tau$, then $|\Delta|; \|\Gamma\| \vdash^i M:\|\tau\|$;
- (b) if $|\Delta|; \|\Gamma\| \vdash^i M:A$, then $|\Delta|; \|\Gamma\| \vdash^r e:\tau$ with $\|\tau\| = A$.

(2) If $|e| = M$, then:

- (a) if $\Delta \vdash^c e:\sigma$, then $|\Delta| \vdash^i M:|\sigma|$;
- (b) if $|\Delta| \vdash^i M:A$, then $\Delta \vdash^c e:\sigma$ with $|\sigma| = A$.

PROOF. By simultaneous induction on the definitions of $\|e\|$ and $|e|$. Note that we can take advantage of strong inversion properties, since we have exactly one typing rule for each term constructor in Mini-ML[□] and Mini-ML₂, plus the up and down rules to connect the \vdash^c and \vdash^r judgments.

We only show the two cases involving the up and down rules since all others are easy.

Case: $\|\bar{e}\| = \mathbf{unbox}_1 \|\bar{e}\|$, part 1a.

$\Delta; \Gamma \vdash \bar{e} : \tau$	assumption
$\Delta \vdash^c \bar{e} : \tau$	by inversion (rule down)
$ \Delta \vdash^i \ \bar{e}\ : \Box \ \tau\ $	by i.h. 2a
$ \Delta ; \ \Gamma\ \vdash^i \mathbf{unbox}_1 \ \bar{e}\ : \ \tau\ $	by rule tpi_unbox

Case: $\|\bar{e}\| = \mathbf{unbox}_1 \|\bar{e}\|$, part 1b.

$ \Delta ; \ \Gamma\ \vdash^i \mathbf{unbox}_1 \ \bar{e}\ : \ \tau\ $	assumption
$ \Delta \vdash^i \ \bar{e}\ : \Box \ \tau\ $	by inversion (rule tpi_unbox)
$\Delta \vdash^c \bar{e} : \tau$	by i.h. 2b
$\Delta; \Gamma \vdash \bar{e} : \tau$	by rule down

Case: $\|e\| = \mathbf{box} \|e\|$, part 2a.

$\Delta \vdash^c e : \tau$	assumption
$\Delta; \cdot \vdash^c e : \tau$	by inversion (rule up)
$ \Delta ; \cdot \vdash^i \ e\ : \ \tau\ $	by i.h. 1a
$ \Delta \vdash^i \mathbf{box} \ e\ : \Box \ \tau\ $	by rule tpi_box

Case: $\|e\| = \mathbf{box} \|e\|$, part 2b.

$ \Delta \vdash^i \mathbf{box} \ e\ : \Box \ \tau\ $	assumption
$ \Delta ; \cdot \vdash^i \ e\ : \ \tau\ $	by inversion (rule tpi_box)
$\Delta; \cdot \vdash^c e : \tau$	by i.h. 1b
$\Delta \vdash^c e : \tau$	by rule up

□

The translation and proof can be easily generalized from a two-level language to a B-level language [Nielson and Nielson 1992] with an infinite linear ordering. In this case the image of the translation on well-typed terms is exactly the fragment Mini-ML_K^\Box , where \mathbf{unbox}_n is restricted to $n = 1$. This fragment corresponds to a weaker modal logic, K, in which we drop the assumption in S4 that the accessibility relation is reflexive and transitive [Martini and Masini 1996], and which we discussed briefly in Section 2. Thus a corollary of the generalized theorem is that Mini-ML_K^\Box is equivalent to an infinite linear B-level language, since the translation is then a bijection which preserves correctness of typing. If we further restrict Mini-ML_K^\Box so that rule tpi_box is only allowed when we have exactly one context in the context stack, then we obtain an exact correspondence with the two-level language, but this does not seem correspond to a previously studied logic.

7. Examples

We now present some standard examples from partial evaluation to illustrate the expressiveness of our language Mini-ML^\Box . We use $\mathbf{let} x = E_1 \mathbf{in} E_2$ to introduce (nonpolymorphic) top-level definitions; it may be considered syntactic sugar for $(\lambda x:A. E_2)E_1$.

7.1. ACKERMANN'S FUNCTION. We now present a program for calculating Ackermann's function that specializes to the first argument. It is based on the following program:

```

fix acker:nat → nat → nat.
  λm:nat. case m
    of z ⇒ λn:nat. sn
    |sm' ⇒ λn:nat. case n
      of z ⇒ acker m' (sz)
      |sn' ⇒ acker m' (acker mn').

```

Now, if we attempt to directly insert the modal constructors to divide this program into two stages, we get the following:

```

fix acker:nat → □(nat → nat).
  λm:nat. case m
    of z ⇒ box(λn:nat. sn)
    |sm' ⇒ box(λn:nat. case n
      of z ⇒ (unbox1(acker m'))(sz)
      |sn' ⇒ (unbox1(acker m'))((unbox1(acker m))n').

```

Unfortunately, when applied to the first argument, this function generally will not terminate. This is a common problem in partial evaluation, and the usual solution is to employ memoization during specialization, which works for many programs. Here we will simply note that the problem in this case is a recursive call to *acker m* while calculating *acker m*, which can be removed by adding an additional fixpoint as follows.

```

fix acker:nat → □(nat → nat).
  λm:nat. case m
    of z ⇒ box(λn:nat. sn)
    |sm' ⇒ box(fix ackm. λn:nat.
      case n
        of z ⇒ (unbox1(acker m'))(sz)
        |sn' ⇒ (unbox1(acker m'))(ackm n').

```

This function will always terminate. The recursive applications appearing inside **unbox**₁ constructors are evaluated when the first argument is given. The compilation of this function to Mini-ML_e[□] makes this more explicit:

```

fix acker:nat → □(nat → nat).
  λm:nat. case m
    of z ⇒ box(λn:nat. sn)
    |sm' ⇒ let box f = acker m' in
      let box g = acker m' in
      box(fix ackm. λn:nat.
        case n of z ⇒ f(sz)
        |sn' ⇒ g(ackm n').

```

```

let times':  $\Box(\text{nat} \rightarrow \Box(\text{nat} \rightarrow \text{nat})) =$ 
  box ( $\lambda m:\text{nat}.$  case  $m$ 
    of  $z \Rightarrow$  box ( $\lambda n:\text{nat}.$   $z$ )
    |  $s \ m' \Rightarrow$  box ( $\lambda n:\text{nat}.$  times  $n$  (unbox1 (liftnat  $m$ ))))
in let iprod' = fix ip:  $\text{nat} \rightarrow \Box(\text{vector} \rightarrow \Box(\text{vector} \rightarrow \text{nat}))$ .
   $\lambda n:\text{nat}.$  case  $n$ 
    of  $z \Rightarrow$  box ( $\lambda v:\text{vector}.$  box ( $\lambda w:\text{vector}.$   $z$ ))
    |  $s \ n' \Rightarrow$  box ( $\lambda v:\text{vector}.$  box ( $\lambda w:\text{vector}.$ 
      plus (unbox1 (unbox1 times' (unbox1 (sub'  $n$ )  $v$ ))
        (unbox2 (sub'  $n$ )  $w$ ))
        (unbox1 (unbox1 (ip  $n'$ )  $v$ )  $w$ )))
in let iprod3 :  $\text{vector} \rightarrow \Box(\text{vector} \rightarrow \text{nat}) =$  unbox0(iprod' 3)
in let iprod3a :  $\text{vector} \rightarrow \text{nat} =$  unbox0 (iprod3 [7, 0, 9])
in let iprod3b :  $\text{vector} \rightarrow \text{nat} =$  unbox0 (iprod3 [7, 8, 0])
in ...

```

FIG. 1. Staged code for inner product.

Notice that *acker* m' is unnecessarily calculated twice. This would be avoided if memoization was employed during the compilation or if we had explicitly bound a variable to the result of this computation.

7.2. INNER PRODUCTS. In Glück and Jørgensen [1995], the calculation of inner products is given as an example of a program with more than two phases. We now show how this example can be coded in Mini-ML[□]. We assume a data type *vector* in the example, along with a function $\text{sub}:\text{nat} \rightarrow \text{vector} \rightarrow \text{nat}$ to access the elements of a *vector*.

Then, the inner product example without staging is expressed in Mini-ML as follows:

```

fix ip:  $\text{nat} \rightarrow \text{vector} \rightarrow \text{vector} \rightarrow \text{nat}.$ 
 $\lambda n:\text{nat}.$  case  $n$ 
  of  $z \Rightarrow \lambda v:\text{vector}.$   $\lambda w:\text{vector}.$   $z$ 
  |  $sn' \Rightarrow \lambda v:\text{vector}.$   $\lambda w:\text{vector}.$ 
    plus (times(sub  $n$   $v$ ) (sub  $n$   $w$ ))
    (ip  $n'$   $v$   $w$ )

```

We add in \Box , **box** and **unbox_i** to obtain a function with three computation stages which is shown in Figure 1. We assume a function lift_{nat} as defined earlier and a function $\text{sub}':\text{nat} \rightarrow \Box(\text{vector} \rightarrow \text{nat})$ which is a specializing version of *sub*, that perhaps precomputes some pointer arithmetic based on the array index. We first define a staged version *times'* of *times* which avoids the multiplication in the specialization if the first argument is zero. This will speed up application of *iprod'* to its third argument, particularly in the case that the second argument is a sparse vector.

The last three lines show how to execute the result of a specialization using **unbox₀** (corresponding to *eval* in Lisp). Also, the occurrence of **unbox₂** indicates code used at the third stage but generated at the first. These two aspects could not be expressed within the multi-level language in Glück and Jørgensen [1995].

Note the erasure of the **unbox_i** and **box** constructors in *iprod'* leaves the unstaged code, except that we used a different version of multiplication. The operational semantics of the two programs is of course quite different.

```

(* val acc : regexp -> (string list -> bool) -> (string list -> bool) *)
fun acc (Empty) k s = k s
  | acc (Plus(r1,r2)) k s = acc r1 k s orelse
                           acc r2 k s
  | acc (Times(r1,r2)) k s =
    acc r1 (fn ss => acc r2 k ss) s
  | acc (Star(r)) k s =
    k s orelse
    acc r (fn ss => if s = ss then false
                  else acc (Star(r)) k ss) s
  | acc (Const(str)) k (x::s) =
    (x = str) andalso k s
  | acc (Const(str)) k (nil) = false

(* val accept : regexp -> (string list -> bool) *)
fun accept r s =
  acc r (fn nil => true | (x::l) => false) s

```

FIG. 2. Unstaged regular expression matcher.

7.3. REGULAR EXPRESSION MATCHING. We now present a program for regular expression matching that specializes to a particular regular expression. We use the full Standard ML language, augmented with our modal constructors. Our program is based on the nonspecializing one in Figure 2, which makes use of a continuation function that is called with the remaining input if the current matching succeeds. We assume the following datatype declaration:

```

datatype regexp
  = Empty
  | Plus of regexp * regexp
  | Times of regexp * regexp
  | Star of regexp
  | Const of string

```

Note that there is a recursive call to `acc (Star(r))` in the case for `acc (Star(r))` which we can transform using a local definition, similar to the **fix** introduced in the Ackermann function example. This must be done so that specialization with respect to the regular expression terminates. The resulting code for this case is:

```

  | acc (Star(r)) k s =
    let fun accStar k s =
        k s orelse
        acc r
          (fn ss => if s = ss then false
                  else accStar k ss)
      in
        s
      end
    in
      accStar k s
    end

```

```

(* val acc2 : regexp -> [(string list -> bool) -> (string list -> bool)] *)
fun acc2 (Empty) = ' fn k => fn s => k s
  | acc2 (Plus(r1,r2)) = ' fn k => fn s =>
    ^ (acc2 r1) k s orelse
    ^ (acc2 r2) k s
  | acc2 (Times(r1,r2)) = ' fn k => fn s =>
    ^ (acc2 r1) (fn ss => ^ (acc2 r2) k ss) s
  | acc2 (Star(r)) = ' fn k => fn s =>
    let fun acc2Star k s =
      k s orelse
      ^ (acc2 r)
        (fn ss => if s = ss then false
                  else acc2Star k ss)
    in
      s
    in
      acc2Star k s
    end
  | acc2 (Const(str)) = ' fn k =>
    (fn (x::ss) =>
      (x = ^ (lift_string str))
      andalso k ss
    | nil => false)

(* val accept2 : regexp -> [string list -> bool] *)
fun accept2 r = ' fn s =>
  ^ (acc2 r) (fn nil => true | (x::l) => false) s

```

FIG. 3. Modally staged regular expression matcher.

Then, we can add in modal constructors to get the staged program in Figure 3 with the following types (using $[A]$ here to represent $\Box A$, following the syntax of PML [Wickline et al. 1998a])

```

val acc2: regexp → [(string list → bool) → (string
list → bool)]
val accept2: regexp → [string list → bool]

```

These types indicate that the required staging is achieved by the program. Inserting the modal constructors requires breaking up the function arguments, but is otherwise relatively straightforward. We use \cdot for **box** and \wedge for **unbox**₁. More generally, we suggest using \wedge_n for **unbox**_n.

We can now use our compilation to the explicit language Mini-ML_e[□] to get an equivalently staged program. We can then further translate to a program in pure Standard ML, which is staged in the same way, but without the modal annotations, as shown in Figure 4. It is unnecessary to replace $[A]$ by $\text{unit} \rightarrow A$ in this case, since the code constructor (\cdot) is only applied to values. We show this program only to demonstrate the staging described by the modal annotated program. The program in Mini-ML_e[□] has the potential to be more efficient, since optimized code can be generated by a sophisticated implementation.

```

(* val acc3 : regexp -> (string list -> bool) -> (string list -> bool) *)
fun acc3 (Empty) = (fn k => fn s => k s)
  | acc3 (Plus(r1,r2)) =
    let val a1 = acc3 r1
        val a2 = acc3 r2
    in
      (fn k => fn s => a1 k s orelse a2 k s)
    end
  | acc3 (Times(r1,r2)) =
    let val a1 = acc3 r1
        val a2 = acc3 r2
    in
      (fn k => fn s => a1 (fn ss => a2 k ss) s)
    end
  | acc3 (Star(r1)) =
    let val a1 = acc3 r1
        fun acc3Star k s =
          k s orelse
            a1 (fn ss => if s = ss then false
                        else acc3Star k ss)
    in
      (fn k => fn s => acc3Star k s)
    end
  | acc3 (Const(str)) =
    (fn k => (fn (x::s) => (x = str) andalso k s)
      | nil => false))

(* val accept3 : regexp -> (string list -> bool) *)
fun accept3 r =
  acc3 r (fn nil => true | (x::l) => false)

```

FIG. 4. Pure SML staged regular expression matcher.

8. Related Work

Our modal $\lambda_e^{\rightarrow\Box}$ -calculus is originally based on the modal λ -calculus presented by Bierman and de Paiva [1996] and used by Pfenning and Wong [1995], who call it the “explicit system.” Our calculus avoids the use of simultaneous substitution by using both a modal context and an ordinary one, following the sequent calculi proposed by Andreoli [1992] for linear logic and by Girard [1993] for LU. The result is similar to the linear λ -calculus formulated by Wadler [1993]. In a separate work reported in Pfenning and Davies [2001] we show how our calculus and associated natural deduction system can be carefully constructed following the methodology of Martin-Löf [1996] and also show how to account for modal possibility using two forms of conclusion.

The language Mini-ML $_{\Box}^{\rightarrow}$ is constructed by combining $\lambda_e^{\rightarrow\Box}$ and Mini-ML [Clément et al. 1986]. The language Mini-ML $_{\Box}^{\rightarrow}$ is based on the “implicit” modal λ -calculus presented in Pfenning and Wong [1995], which uses a stack of ordinary contexts rather than two contexts. Mini-ML $_{\Box}^{\rightarrow}$ avoids the pop structural rule of

Pfenning and Wong [1995], which is difficult to motivate from the point of view of natural deduction, by instead removing contexts from the stack at the $\Box E$ rule. The compilation from Mini-ML^\Box to Mini-ML_e^\Box is inspired by one direction of the proof of equivalence between the two calculi given in Pfenning and Wong [1995]. Systems similar to the implicit modal λ -calculus of Pfenning and Wong [1995] have been proposed by Martini and Masini [1996], who introduce a simple reduction semantics, and Bourghuis [1994], who considers modal pure type systems. None of the prior work on modal λ -calculi has considered the relationship to computation staging.

Partly motivated by a previous version of the current paper [Davies and Pfenning 1996], Goubault-Larrecq [1996; 1997] has proposed a formulation of modal λ -calculi using explicit substitutions. While this system has some interesting properties as a calculus, in particular giving a finer grained analysis of reduction and equality, it is unclear how this is relevant to the design of a programming language with staging primitives.

Despite some superficial similarities, our code types are quite different from Moggi's computational types based on monads [Moggi 1989; 1991], which only distinguish values from computations and do not allow expression of stage separation. Moreover, our intended implementation of code is *intensional*, since we wish to allow refinements of our semantics to optimize code, while Moggi's computations are *extensional* with evaluation as the only operation. In other work [Pfenning and Davies 2001], we have been able to explain computational types cleanly in our framework via a combination of the intuitionistic possibility operator \Diamond and necessity. This follows an earlier suggestion by Kobayashi [1997] and a related investigation by Benton et al. [1998] who establish a connection between the computational λ -calculus and lax logic [Fairtlough and Mendler 1997].

Another application of a modal λ -calculus to capture intensionality is reported in Despeyroux et al. [1997]. This differs from our application in that it provides a primitive recursion construct for examining terms with types of the form $\Box A$. It also captures extensionality as given by the local expansions, thus obtaining a calculus suitable as a meta-logical framework, that is, a logical framework in which we can reason about the specified logics. The modal operator here serves to avoid the usual paradoxes that would arise if we incorrectly identify an expression with its denotation.

A further application of modal λ -calculi is to capture complexity classes such as polynomial-time computation [Hofmann 1998; 2000] by restricting recursion to terms with types of the form $\Box A$.

We have shown how some standard examples of specialization can be expressed in Mini-ML^\Box . More complicated examples might require more advanced techniques to achieve the desired staging, such as the binding-time improvements used in partial evaluation. Memoizing when generating code is another useful technique used in partial evaluation, and Wickline et al. [1998b] shows how this technique can be programmed in a language with modal types. In general, we expect that any program that can be successfully partially evaluated should be able to be manually staged by the programmer in a language based on Mini-ML^\Box , although more experiments need to be conducted to determine how practical this is for large programs. See Birkedal and Welinder [1993] for a

description of a realistic partial evaluator for Standard ML and Jones et al. [1993] for an overview of standard techniques and examples of partial evaluation.

One possible criticism of our languages is that they only manipulate *closed* code during execution, which restricts the staging that can be expressed compared to the two-level languages used in partial evaluation such as that proposed by Gomard and Jones [1991]. This is solved in Mini-ML[□] by λ -abstracting code expressions over their free variables, and then later generating an application to the actual variables. This results in a number of variable for variable β -redices in the generated code in our examples. Of course, a lower-level implementation could reduce these redices for efficiency, but this is not reflected in the language semantics. From a practical point of view, the reason for only treating closed code is that we need to be able to evaluate code without danger of encountering unbound variables. This is in contrast to the binding-time languages used in partial evaluation, which allow manipulation of code containing free variables, but do not support evaluation of code as a construct within the language. Instead, evaluation of the result of partial evaluation is an external operation applied only to whole programs, the properties of which have been studied separately by Jones [1992].

In other work [Davies 1996], one of the present authors has shown that the \bigcirc (“next”) operator from non-branching temporal logic exactly models the looser correctness criterion used in partial evaluation. Interestingly, the resulting languages are unsound when general references or value carrying exceptions are added, since these features allow a code expression with free variables to escape the binders for those variables. The language with \bigcirc corresponds closely to Mini-ML[□]_K (see Section 6.4) except that it has the looser correctness criterion. Logically, \square in Mini-ML[□]_K exactly corresponds to \bigcirc in *branching* temporal logic, so the looser criterion is exactly captured by the move from branching to non-branching temporal logic.

Taha and Sheard [1997] have directly constructed a language similar to Mini-ML[□], which allows manipulation of code with free variables as well as type-safe evaluation, but their original design proved to be unsound in that free variables may be encountered during evaluation. This is fixed in Taha et al. [1998], resulting in a language called Meta-ML, which is sound in the absence of references and exceptions, but it seems more operationally, rather than logically motivated. More recent work on Meta-ML has concentrated on an idealized language [Moggi et al. 1999] and makes direct use of the results in the previous version of this paper [Davies and Pfenning 1996] as well as Davies [1996].

Over the last few years, there has been a lot of interest in run-time code generation in high-level languages. Engler et al. [1996] describe an extension of the programming language C called ‘C (pronounced “tick C”’), which uses similar mechanisms to Mini-ML[□] to achieve computation staging. However, the type system lacks the modal restriction on variables, so it allows variables to be used when their values are not available, which may result in incorrect results or runtime errors. Consel and Noël [1996] describe a system called Tempo which allows both partial evaluation and run-time code generation for the C language. Standard binding-time analysis techniques are used, along with separate annotations to describe where run-time specialization should be done. These annotations roughly correspond to the **unbox**₀ construct in Mini-ML[□], while the annotations resulting from the binding-time analysis roughly correspond to the

box and **unbox**₁ constructs, although the restriction of Mini-ML[□] to closed code means that this correspondence is not exact. Leone and Lee [Leone and Lee 1994; Lee and Leone 1996] describe a small ML-like language and a corresponding implementation called Fabius which treats curried functions as run-time code generators, using a new form of binding-time analysis. The staging achieved is quite different to that obtained using ordinary binding-time analysis, and one of the original motivations for the current work was to allow a formal characterization of this staging. Fabius also uses a very fast form of run-time code generation called deferred compilation.

Deferred compilation has also recently been used by Wickline et al. [1998a] as an implementation technique for a language based on Mini-ML_e[□] called PML, which includes most of core SML and performs run-time code generation based on modal types. The extension to core SML was very smooth even though the language includes polymorphism, datatypes, references and arrays. A simple compiler for this language has been completed, and work is continuing on improved implementations.

9. Conclusion and Future Work

In this paper, we have proposed a logical interpretation of binding times and staged computation in terms of the intuitionistic modal logic S4. We first presented the $\lambda_e^{\rightarrow\Box}$ -calculus, and formally demonstrated the sense in which it captures staging. We then extended this to the explicit language Mini-ML_e[□] (including recursion, natural numbers, and products) and presented its natural operational semantics. We continued by defining an implicit language Mini-ML[□] which might serve as the core for an extension of a language with the complexity of Standard ML, and which is syntactically similar to both Lisp's backquote and comma notation, as well as the languages used in partial evaluation. The operational semantics of Mini-ML[□] is given by a type-preserving compilation to Mini-ML_e[□]. Further, Mini-ML[□] generalizes Nielson and Nielson's two-level functional language [Nielson and Nielson 1992] which is demonstrated by a conservative embedding theorem, an important technical result of this paper.

Our approach provides a general, logically motivated framework for staged computation that includes aspects of both partial evaluation and run-time code generation. As such it allows efficient code to be generated within a declarative style of programming, and provides an automatic check that the intended staging is achieved.

Our investigation remains at a relatively abstract level, thus providing a general framework in which various staging mechanisms may be studied from a new point of view. We implemented the original interpreter for Mini-ML[□] in the logic programming language Elf [Pfenning 1991], thus allowing us to perform small experiments at this abstract level. Concrete instances such as partial evaluation, run-time code generation, or macro expansion will require some additional considerations for their effective use and efficient implementation. The application to run-time code generation appears particularly promising and is described in more detail, including an extended example, in work by the current authors in conjunction with Wickline et al. [1998b]. We hope that future design and implementation work will lead to a practical full-scale programming language with computation staging based on modal types.

ACKNOWLEDGMENTS. The work described here has benefited from discussions with too many colleagues to acknowledge them here individually. We would like to particularly thank Olivier Danvy, Peter Lee, Philip Wickline and the anonymous referees for numerous fruitful discussions and suggestions.

REFERENCES

- ANDREOLI, J.-M. 1992. Logic programming with focusing proofs in linear logic. *J. Logic Comput.* 2, 3, 197–347.
- BENTON, P. N., BIERMAN, G. M., AND DE PAIVA, V. C. V. 1998. Computational types from a logical perspective. *J. Funct. Prog.* 8, 2 (Mar.), 177–193.
- BIERMAN, G., AND DE PAIVA, V. 1996. Intuitionistic necessity revisited. Tech. Rep. CSRP-96-10 (June). School of Computer Science, Univ. Birmingham, Birmingham, Al.
- BIRKEDAL, L., AND WELINDER, M. 1993. Partial evaluation of Standard ML. Master's thesis. Dept. Comput. Sci., Univ. Copenhagen, Copenhagen, Denmark. Available as Tech. Rep. DIKU-report 93/22.
- BORGHUIS, T. 1994. Coming to terms with modal logic: On the interpretation of modalities in typed λ -calculus. Ph.D. dissertation. Eindhoven University of Technology, Eindhoven, The Netherlands.
- CHELLAS, B. 1980. *Modal Logic: An Introduction*. Cambridge University Press, Cambridge, Mass., and New York.
- CLÉMENT, D., DESPEYROUX, J., DESPEYROUX, T., AND KAHN, G. 1986. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming (LFP'86)* (Cambridge, Mass., Aug.). ACM, New York, pp. 13–27.
- CONSEL, C., AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)* (St. Petersburg Beach, Fla., Jan. 21–24). ACM, New York, pp. 145–156.
- DAVIES, R. 1996. A temporal-logic approach to binding-time analysis. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS'96)* E. Clarke Ed., (July). IEEE Computer Society Press, Los Alamitos, Calif., pp. 184–195.
- DAVIES, R., AND PFENNING, F. 1996. A modal analysis of staged computation. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)* (St. Petersburg Beach, Fla., Jan. 21–24). ACM, New York, pp. 258–270.
- DESPEYROUX, J., PFENNING, F., AND SCHÜRMANN, C. 1997. Primitive recursion for higher-order abstract syntax. In *Proceedings of the 3rd International Conference on Typed Lambda Calculus and Applications (TLCA'97)* (Nancy, France, Apr.), R. Hindley Ed., Lecture Notes in Computer Science, vol. 1210. Springer-Verlag, New York, pp. 147–163. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon Univ., Pittsburgh, Pa.
- ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 1996. C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)* (St. Petersburg Beach, Fla., Jan. 21–24). ACM, New York, pp. 131–144.
- FAIRTLOUGH, M., AND MENDLER, M. 1997. Propositional lax logic. *Inf. Comput.* 137, 1 (Aug.), 1–33.
- GIRARD, J.-Y. 1993. On the unity of logic. *Ann. Pure Appl. Logic* 59, 201–217.
- GLÜCK, R., AND JØRGENSEN, J. 1995. Efficient multi-level generating extensions for program specialization. In *Proceedings of the 7th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95)* (Sept.). S. Swierstra and M. Hermenegildo, eds. Lecture Notes in Computer Science, vol. 982. Springer-Verlag, New York, pp. 259–278.
- GOMARD, C., AND JONES, N. 1991. A partial evaluator for the untyped lambda-calculus. *J. Funct. Prog.* 1, 1 (Jan.), 21–69.
- GOUBAULT-LARRECO, J. 1996. On computational interpretations of the modal logic S4, parts I–III. Tech. Rep. 1996-33,34,35. Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Karlsruhe, Germany.
- GOUBAULT-LARRECO, J. 1997. On computational interpretations of the modal logic S4-IIIb. confluence, termination of the $\lambda_{\text{ev}}Q_H$ -calculus. Tech. Rep. 3164 (May). INRIA, France.

- HATCLIFF, J. 1995. Mechanically verifying the correctness of an offline partial evaluator. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'95)* (Utrecht, The Netherlands, Sept.). M. Hermenegildo and S. Swierstra, eds. Lecture Notes in Computer Science, vol. 982. Springer-Verlag, New York, pp. 279–298.
- HOFMANN, M. 1998. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *Computer Science Logic (CSL'97): Annual Conference of the EACSL (Selected Papers)* (Aarhus, Denmark, Aug.). M. Nielsen and W. Thomas, eds. Lecture Notes in Computer Science, vol. 1414. Springer-Verlag, New York, pp. 275–294.
- HOFMANN, M. 2000. Programming languages capturing complexity classes. *ACM SIGACT News* 31, 1 (Mar.), 31–42.
- JONES, N. D. 1992. Efficient algebraic operations on programs. In *Algebraic Methodology and Software Technology (AMAST'91): Proceedings of the Second International Conference*. M. Nivat, C. Rattray, T. Rus, and G. Scollo, eds. Springer, London, England, pp. 393–420. (A version appears as a chapter in Jones et al. [1993].)
- JONES, N. D., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, International Series in Computer Science, Englewood Cliffs, N.J.
- JØRRING, U., AND SCHERLIS, W. L. 1986. Compilers and staging transformations. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL'86)* (St. Petersburg Beach, Fla., Jan.). ACM, New York, pp. 86–96.
- KEPPEL, D., EGGERS, S. J., AND HENRY, R. R. 1993. A case for runtime code generation. Tech. Rep. TR 93-11-02 (Nov.). Dept. Comput. Sci. Eng., Univ. Washington, Seattle, Wash.
- KOBAYASHI, S. 1997. Monad as modality. *Theoret. Comput. Sci.* 175, 29–74.
- KRIPKE, S. A. 1963. Semantic analysis of modal logic. I: Normal propositional calculi. *Zeitsc. Math. Logik Grund. Math.* 9, 67–96.
- LEE, P., AND LEONE, M. 1996. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)* (Philadelphia, Pa., May 21–24). ACM, New York, pp. 137–148.
- LEONE, M., AND LEE, P. 1994. Lightweight run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'94)* (Orlando, Fla., June). ACM, New York.
- MARTIN-LÖF, P. 1996. On the meanings of the logical constants and the justifications of the logical laws. *Nordic J. Philos. Logic* 1, 1, 11–60.
- MARTINI, S., AND MASINI, A. 1996. A computational interpretation of modal proofs. In *Proof Theory of Modal Logics*, H. Wansing, ed. Kluwer, Dordrecht, The Netherlands, pp. 213–241.
- MOGGI, E. 1989. Computational lambda calculus and monads. In *Proceedings of the 4th Symposium on Logic in Computer Science (LICS'89)* (Asilomar, Calif., June). IEEE Computer Society Press, Los Alamitos, Calif., pp. 14–23.
- MOGGI, E. 1991. Notions of computation and monads. *Inf. Comput.* 93, 1, 55–92.
- MOGGI, E., TAHA, W., BENAÏSSA, Z.-E.-A., AND SHEARD, T. 1999. An idealized MetaML: Simpler, and more expressive. In *Proceedings of the European Symposium on Programming* (Amsterdam, The Netherlands, Mar.), Lecture Notes in Computer Science, vol. 1576. Springer-Verlag, New York, pp. 193–207.
- NIELSON, F., AND NIELSON, H. R. 1992. *Two-Level Functional Languages*. Cambridge University Press, Cambridge, Mass.
- PALSBERG, J. 1993. Correctness of binding time analysis. *J. Funct. Prog.* 3, 3 (July), 347–363.
- PFENNING, F. 1991. Logic programming in the LF logical framework. In *Logical Frameworks*. G. Huet and G. Plotkin, eds., Cambridge University Press, Cambridge, Mass., pp. 149–181.
- PFENNING, F., AND DAVIES, R. 2001. A judgmental reconstruction of modal logic. *Math. Struct. Comput. Sci.* To appear.
- PFENNING, F., AND WONG, H.-C. 1995. On a modal λ -calculus for S4. In *Proceedings of the 11th Conference on Mathematical Foundations of Programming Semantics (MFPS'95)* (New Orleans, La., Mar.). S. Brookes and M. Main, eds. Available as *Electronic Notes in Theoretical Computer Science*, Vol. 1. Elsevier, Amsterdam, The Netherlands.
- PRAWITZ, D. 1965. *Natural Deduction*. Almqvist & Wiksell, Stockholm, Sweden.
- SMITH, B. C. 1984. Reflection and semantics in LISP. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL'84)* (Salt Lake City, Ut., Jan.). ACM, New York, pp. 23–35.

- TAHA, W., BENAÏSSA, Z.-E.-A., AND SHEARD, T. 1998. Multi-stage programming: Axiomatization and type safety. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP'98)* (July), Lecture Notes in Computer Science, vol. 1443. Springer-Verlag, New York, pp. 918–929.
- TAHA, W., AND SHEARD, T. 1997. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)* (Amsterdam, The Netherlands, June 12–13). ACM, New York, pp. 203–217.
- WADLER, P. 1993. A taste of linear logic. In *Mathematical Foundations of Computing Science (MFCS'93)* (Gdansk, Poland, Aug.). Lecture Notes in Computer Science, vol. 711. Springer-Verlag, New York, pp. 185–210.
- WICKLINE, P., LEE, P., AND PFENNING, F. 1998a. Modal types as staging specifications for run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)* (Montreal, Ont. Canada, June 17–19). K. D. Cooper, Ed. ACM, New York, pp. 224–235.
- WICKLINE, P., LEE, P., PFENNING, F., AND DAVIES, R. 1998b. Modal types as staging specifications for run-time code generation. *ACM Comput. Surv.* 30, (Sept.), 3es.

RECEIVED JULY 1999; REVISED AUGUST 2000; ACCEPTED AUGUST 2000