A Modal Analysis of Staged Computation

Rowan Davies and Frank Pfenning

August 1999

CMU-CS-99-153

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as FOX Memorandum CMU-CS-FOX-99-02

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 lambda-calculi and functional languages. We directly demonstrate the sense in which our calculus captures staging, and also give a conservative embedding of Nielson & Nielson's two-level functional language in our language, thus proving that binding-time correctness is equivalent to modal correctness. In addition, our language can express immediate evaluation and sharing of code across multiple stages, thus supporting run-time code generation as well as partial evaluation.
This is an extended and revised version of the conference paper [DP96].

This work was sponsored in part by the National Science Foundation under grant CCR-9619832 and by the Advanced Research Projects Agency (ARPA) CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628–91–C–0168. This work was also partly supported by a Hackett Studentship from the University of Western Australia. Part of this work was completed during a visit by the first author to BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation)

Keywords: Programming Languages, Staged Computation, Modal Logic

Contents

1 Introduction

Dividing a computation into separate stages is a common informal technique for the derivation of algorithms [JS86]. 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, for example, [JGS93]) divides the computation into two stages based on the early availability of some function arguments. In practice this appears most successful when supported by *binding-time analysis* [GJ91], 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 [NN92] and some work on partial evaluation (for example, [GJ91]) 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 [Pra65]. 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 [Jon91] 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 [WLPD98, WLP98].

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 $B \to \Box A$ can carry out all computation concerned with its argument while generating the residual code of type A.

We begin by considering $\lambda_e^{\to\Box}$, a modal λ-calculus based on a natural-deduction formulation of intuitionistic modal S4. The presentation is new, but draws on ideas in [BdP92, PW95, Gir93].

We then construct a functional language Mini-ML$_e^\Box$ by augmenting $\lambda_e^{\to\Box}$ 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 [CDDK86].

Mini-ML$_e^\Box$ 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$_e^\Box$, but it also makes it difficult to directly relate it to previous work on staged languages, such as two-level languages [NN92]. We thus consider a more implicit formulation of S4 motivated by its Kripke semantics following [MM94, PW95] and then augment it as before to form Mini-ML$^{\Box}$. With some syntactic sugar, Mini-ML$^{\Box}$ 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$^{\Box}$ is given by a type-preserving translation to Mini-ML$^{\Box}_{e}$ whose correctness is not entirely trivial. This translation also describes the first phase of a plausible compilation strategy for Mini-ML$^{\Box}$ for run-time code generation.

We then exhibit a simple full and faithful embedding of Nielson & Nielson's two-level language [NN92] in Mini-ML$^{\Box}$, providing further evidence that Mini-ML$^{\Box}$ 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 $\lambda_e^{\rightarrow\Box}$. 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 $\Box$ operator, and show that they satisfy subject reduction and expansion, respectively. We also demonstrate the relationship between $\lambda_e^{\rightarrow\Box}$ and computation staging via two theorems.

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 $\Box$, where $\Box A$ 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** $E : \Box A$ 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 [Pra65]: We can infer that **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 suggest *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 (see [PW95] for a counterexample). 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 proposition are true and others are valid. We write

$$(A_1, \ldots, A_m); (B_1, \ldots, B_n) \vdash^{e} C$$

to express

Under the hypothesis that $A_1, \ldots, A_m$ are valid and $B_1, \ldots, B_n$ 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.

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

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

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

$$(A_1, \ldots, A_m); \cdot \vdash^{e} C$$

or, with proof terms,

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

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

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

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.

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

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 corrspondence). We want to express that $A \to B$ should be true if B is true under the hypothesis that A is true.

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

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

Conversely, if we know that $A \to B$ 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.

$$\frac{\Delta; \Gamma \vdash^{e} E_2 : A \to B \qquad \Delta; \Gamma \vdash^{e} E_1 : A}{\Delta; \Gamma \vdash^{e} E_2\, E_1 : B} \to\mathsf{E}$$

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 completness 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.

$$\frac{\dfrac{\begin{array}{c}\mathcal{D}_2\\ \Delta; \Gamma, x{:}A \vdash^{e} E_2 : B\end{array}}{\Delta; \Gamma \vdash^{e} (\lambda x{:}A.\ E_2) : A \to B} \to\mathsf{I} \qquad \begin{array}{c}\mathcal{D}_1\\ \Delta; \Gamma \vdash^{e} E_1 : A\end{array}}{\Delta; \Gamma \vdash^{e} (\lambda x{:}A.\ E_2)\, E_1 : B} \to\mathsf{E} \quad \Longrightarrow \quad \begin{array}{c}\mathcal{D}_2'\\ \Delta; \Gamma \vdash^{e} [E_1/x]E_2 : B\end{array}$$

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.

$$\begin{array}{c}\mathcal{D}\\ \Delta;\Gamma \vdash^{e} E : A \to B\end{array} \Longrightarrow \dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ \Delta;\Gamma, x{:}A \vdash^{e} E : A \to B\end{array} \qquad \dfrac{}{\Delta;\Gamma, x{:}A \vdash^{e} x : A}\,\mathsf{ovar}}{\Delta;\Gamma, x{:}A \vdash^{e} E\,x : B}\,{\to}\mathsf{E}}{\Delta;\Gamma \vdash^{e} (\lambda x{:}A.\ E\,x) : A \to B}\,{\to}\mathsf{I}$$

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:

$$\dfrac{\Delta;\cdot \vdash^{e} E : A}{\Delta;\Gamma \vdash^{e} \mathbf{box}\ E : \Box A}\,\Box\mathsf{I}$$

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 which 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.

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

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

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Delta;\cdot \vdash^{e} E_1 : A\end{array}}{\Delta;\Gamma \vdash^{e} \mathbf{box}\ E_1 : \Box A}\,\Box\mathsf{I} \qquad \begin{array}{c}\mathcal{D}_2\\ (\Delta, u{:}A);\Gamma \vdash^{e} E_2 : B\end{array}}{\Delta;\Gamma \vdash^{e} \mathbf{let\ box}\ u = \mathbf{box}\ E_1\ \mathbf{in}\ E_2 : B}\,\Box\mathsf{E} \Longrightarrow \begin{array}{c}\mathcal{D}_2'\\ \Delta;\Gamma \vdash^{e} [E_1/u]E_2 : B\end{array}$$

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

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

$$\begin{array}{c}\mathcal{D}\\ \Delta;\Gamma \vdash^{e} E : \Box A\end{array} \Longrightarrow \dfrac{\begin{array}{c}\mathcal{D}\\ \Delta;\Gamma \vdash^{e} E : \Box A\end{array} \qquad \dfrac{\dfrac{}{(\Delta, u{:}A);\cdot \vdash^{e} u : A}\,\mathsf{mvar}}{(\Delta, u{:}A);\Gamma \vdash^{e} \mathbf{box}\ u : \Box A}\,\Box\mathsf{I}}{\Delta;\Gamma \vdash^{e} \mathbf{let\ box}\ u = E\ \mathbf{in\ box}\ u : \Box A}\,\Box\mathsf{E}$$

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*". We leave the details to a future paper, since it does not directly concern our main objective here.

Our presentation simplifies that of the modal λ-calculus $\lambda_e^{\to\Box}$ from [BdP92, PW95] by eliminating the need for simultaneous substitution while preserving subject reduction. It is inspired by sequent calculi proposed by Andreoli [And92] for linear logic and by Girard [Gir93] for LU. Wadler [Wad93] has formulated a linear λ-calculus with two contexts, which shares some features with our calculus. The methodolgy we followed is due to Martin-Löf [ML85a, ML85b], although we have not seen the 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 $\mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2$. Evaluation of code, certainly one of the most fundamental operations, is then definable by

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

Here, and from now on, we associate $\Box$ more strongly than $\to$ 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 a few other examples of modal propositions and proofs from which the natural deductions can be easily reconstructed.

$$\begin{array}{l} \vdash^e \lambda x{:}\Box(A \to B).\ \lambda y{:}\Box A.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ let\ box}\ v = y\ \mathbf{in\ box}\ (u\, v) \\ \quad : \Box(A \to B) \to (\Box A \to \Box B) \end{array}$$

$$\begin{array}{l} \vdash^e \lambda x{:}\Box A.\ \mathbf{let\ box}\ u = x\ \mathbf{in\ box\ box}\ u \\ \quad : \Box A \to \Box\Box A \end{array}$$

$$\begin{array}{l} \vdash^e \lambda x{:}\Box A.\ \mathbf{let\ box}\ u = x\ \mathbf{in}\ u \\ \quad : \Box A \to A \end{array}$$

Note that the first law holds in all modal logics, while the second and third correspond to reflexivity and transitivity of the accessibility relation between worlds in the Kripke semantics [Kri63] in axiomatic formulations of modal logics.

2.2 Syntax

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

$$\begin{array}{llcl} \text{Types} & A & ::= & A_1 \to A_2 \mid \Box A \\ \text{Terms} & E & ::= & x \mid \lambda x{:}A.\ E \mid E_1\, E_2 \mid \\ & & & u \mid \mathbf{box}\ E \mid \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 \\ \text{Ordinary Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\ \text{Modal Contexts} & \Delta & ::= & \cdot \mid \Delta, u{:}A \end{array}$$

We use A, B for 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

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

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

$$\frac{\Delta; \Gamma \vdash^{e} E_1 : B \rightarrow A \qquad \Delta; \Gamma \vdash^{e} E_2 : B}{\Delta; \Gamma \vdash^{e} E_1\, E_2 : A} \rightarrow \mathsf{E}$$

Modal Fragment

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

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

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

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.

$$\begin{array}{rcl}
(\lambda x{:}A.\ E_2)\ E_1 & \stackrel{\beta}{\longmapsto} & [E_1/x]E_2 \\
\mathbf{let\ box}\ u = \mathbf{box}\ E_1\ \mathbf{in}\ E_2 & \stackrel{\beta}{\longmapsto} & [E_1/u]E_2 \\
E : A \to B & \stackrel{\eta}{\longmapsto} & \lambda x{:}A.\ E\ x \\
E : \Box A & \stackrel{\eta}{\longmapsto} & \mathbf{let\ box}\ u = E\ \mathbf{in\ box}\ u
\end{array}$$

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* $E \stackrel{\beta}{\longmapsto} E'$ *then* $\Delta; \Gamma \vdash^e E' : A$.

2. *If* $\Delta; \Gamma \vdash^e E : A$ *and* $E : A \stackrel{\eta}{\longmapsto} E'$ *then* $\Delta; \Gamma \vdash^e E' : A$.

Proof: In the case of a reduction we first apply inversion 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 $\Box\mathsf{E}$ 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:

$E \longmapsto E' \quad E$ reduces to E'

This judgment is defined by the following rules.

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

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

$$\frac{E \longmapsto E'}{\lambda x{:}A.\ E \longmapsto \lambda x{:}A.\ E'} \textsf{cong\_lam}$$

$$\frac{E_1 \longmapsto E_1'}{E_1E_2 \longmapsto E_1'E_2} \textsf{cong\_app1}$$

$$\frac{E_2 \longmapsto E_2'}{E_1E_2 \longmapsto E_1E_2'} \textsf{cong\_app2}$$

$$\frac{E_1 \longmapsto E_1'}{\mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 \longmapsto \mathbf{let\ box}\ u = E_1'\ \mathbf{in}\ E_2} \textsf{cong\_letbox1}$$

$$\frac{E_2 \longmapsto E_2'}{\mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 \longmapsto \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2'} \textsf{cong\_letbox2}$$

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

Theorem 4 (Subject Reduction with Congruences)
If $\Delta; \Gamma \vDash^e E : A$ *and* $E \longmapsto^* E'$ *then* $\Delta; \Gamma \vDash^e E' : A$.

Proof: By a simple induction on the derivation of $E \longmapsto^* E'$, using subject reduction (Theorem 3) for the base cases. $\Box$

To demonstrate the relationship between this reduction relation and computation staging, we roughly follow the binding-time correctness criteria described by Palsberg [Pal93]. Palsberg presented a modular proof of correctness for binding-time analyses based on two-level languages, such as those studied in [GJ91]. 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 $\mathbf{let\ box}\ u = (\lambda x{:}A.\ E_1')\ \mathbf{in}\ E_2'$.

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 $E \longmapsto E'$. 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 critierion for binding-time correctness.

Theorem 5 (Eliminability) *If* $\cdot;\cdot \vdash^{e} E : \Box A$ *and* $E \longmapsto^{*} E'$ *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 $\mathbf{box}\ E_0'$ for some E_0'. Therefore all subterm occurrences in E' are in the scope of a **box** constructor and hence persistent. $\Box$

Thus, it appears that Palsberg's two properties both follow relatively easily from subject reduction for $\lambda_e^{\to\Box}$. 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 $\lambda_e^{\to\Box}$ with labels.

$$\text{Terms} \quad E \quad ::= \quad \cdots \mid E^{l}$$

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

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

$$\frac{}{E^{l} \longmapsto E}\ \mathsf{unlabel}$$

Recall that there is no congruence rule for **box** so that the rule $\mathsf{unlabel}$ 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 $E_1 \longmapsto^{*} E_1'$ iff $E_2 \longmapsto^{*} E_2'$ 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* $E \longmapsto E'$, *then all persistent subterm occurrences of* E' *are labelled with* l.

Proof: By induction on the derivation of $E \longmapsto E'$.
Case:

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

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:

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

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

$$\frac{E_2 \longmapsto E_2'}{\mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 \longmapsto \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2'} \textsf{cong\_letbox2}$$

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:

$$\frac{}{E^l \longmapsto E} \textsf{unlabel}$$

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

□

Interpreted as a statement about code manipulation during evaluation, this theorems 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:

$$\mathbf{let\ box}\ u = \mathbf{box}\ E\ \mathbf{in}\ u \quad \longmapsto E$$

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 constrast to the languages studied by Palsberg.

Evaluation of code is expressed in $\lambda_e^{\rightarrow\Box}$ by occurrences of code variables u which 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.

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

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$_e^{\Box}$, a language that combines some elements of Mini-ML [CDDK86] with the $\lambda_e^{\rightarrow\Box}$-calculus of the previous section. For the sake of simplicity Mini-ML$_e^{\Box}$ 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

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

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.

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

Functions

$$\frac{x{:}A \text{ in } \Gamma}{\Delta; \Gamma \vDash^{e} x : A}\,\mathsf{tpe\_ovar} \qquad \frac{\Delta; (\Gamma, x{:}A) \vDash^{e} E : B}{\Delta; \Gamma \vDash^{e} \lambda x{:}A.\ E : A \to B}\,\mathsf{tpe\_lam}$$

$$\frac{\Delta; \Gamma \vDash^{e} E_1 : B \to A \qquad \Delta; \Gamma \vDash^{e} E_2 : B}{\Delta; \Gamma \vDash^{e} E_1\, E_2 : A}\,\mathsf{tpe\_app}$$

Code

$$\frac{u{:}A \text{ in } \Delta}{\Delta; \Gamma \vDash^{e} u : A}\,\mathsf{tpe\_mvar} \qquad \frac{\Delta; \cdot \vDash^{e} E : A}{\Delta; \Gamma \vDash^{e} \mathbf{box}\ E : \Box A}\,\mathsf{tpe\_box}$$

$$\frac{\Delta; \Gamma \vDash^{e} E_1 : \Box A \qquad (\Delta, u{:}A); \Gamma \vDash^{e} E_2 : B}{\Delta; \Gamma \vDash^{e} \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 : B}\,\mathsf{tpe\_let\_box}$$

Products

$$\frac{\Delta; \Gamma \vDash^{e} E_1 : A_1 \qquad \Delta; \Gamma \vDash^{e} E_2 : A_2}{\Delta; \Gamma \vDash^{e} \langle E_1, E_2 \rangle : A_1 \times A_2}\,\mathsf{tpe\_pair}$$

$$\frac{\Delta; \Gamma \vDash^{e} E : A_1 \times A_2}{\Delta; \Gamma \vDash^{e} \mathbf{fst}\ E : A_1}\,\mathsf{tpe\_fst} \qquad \frac{\Delta; \Gamma \vDash^{e} E : A_1 \times A_2}{\Delta; \Gamma \vDash^{e} \mathbf{snd}\ E : A_2}\,\mathsf{tpe\_snd}$$

$$\frac{}{\Delta; \Gamma \vDash^{e} \langle\,\rangle : 1}\,\mathsf{tpe\_unit}$$

Natural Numbers

$$\frac{}{\Delta; \Gamma \vDash^{e} \mathbf{z} : \mathsf{nat}}\,\mathsf{tpe\_z} \qquad \frac{\Delta; \Gamma \vDash^{e} E : \mathsf{nat}}{\Delta; \Gamma \vDash^{e} \mathbf{s}\ E : \mathsf{nat}}\,\mathsf{tpe\_s}$$

$$\frac{\Delta; \Gamma \vDash^{e} E_1 : \mathsf{nat} \quad \Delta; \Gamma \vDash^{e} E_2 : A \quad \Delta; (\Gamma, x{:}\mathsf{nat}) \vDash^{e} E_3 : A}{\Delta; \Gamma \vDash^{e} (\mathbf{case}\ E_1\ \mathbf{of}\ \mathbf{z} \Rightarrow E_2 \mid \mathbf{s}\ x \Rightarrow E_3) : A}\,\mathsf{tpe\_case}$$

Recursion

$$\frac{\Delta; \Gamma, x{:}A \vDash^{e} E : A}{\Delta; \Gamma \vDash^{e} \mathbf{fix}\ x{:}A.\ E : A}\,\mathsf{tpe\_fix}$$

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 **box** E, making the least commitment concerning lower-level implementations.

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

We evaluate **let box** $x = E_1$ **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

$$\frac{}{\lambda x{:}A.\ E \hookrightarrow \lambda x{:}A.\ E}\ \mathsf{ev\_lam}$$

$$\frac{E_1 \hookrightarrow \lambda x.\ E_1' \qquad E_2 \hookrightarrow V_2 \qquad [V_2/x]E_1' \hookrightarrow V}{E_1\ E_2 \hookrightarrow V}\ \mathsf{ev\_app}$$

Code

$$\frac{}{\mathbf{box}\ E \hookrightarrow \mathbf{box}\ E}\ \mathsf{ev\_box}$$

$$\frac{E_1 \hookrightarrow \mathbf{box}\ E_1' \quad [E_1'/u]E_2 \hookrightarrow V_2}{\mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 \hookrightarrow V_2}\ \mathsf{ev\_let\_box}$$

Products

$$\frac{E_1 \hookrightarrow V_1 \qquad E_2 \hookrightarrow V_2}{\langle E_1, E_2 \rangle \hookrightarrow \langle V_1, V_2 \rangle}\ \mathsf{ev\_pair}$$

$$\frac{E \hookrightarrow \langle V_1, V_2 \rangle}{\mathbf{fst}\ E \hookrightarrow V_1}\ \mathsf{ev\_fst} \qquad \frac{E \hookrightarrow \langle V_1, V_2 \rangle}{\mathbf{snd}\ E \hookrightarrow V_2}\ \mathsf{ev\_snd}$$

$$\frac{}{\langle\,\rangle \hookrightarrow \langle\,\rangle}\ \mathsf{ev\_unit}$$

Natural Numbers

$$\frac{}{\mathbf{z} \hookrightarrow \mathbf{z}}\,\textsf{ev\_z} \qquad \frac{E \hookrightarrow V}{\mathbf{s}\ E \hookrightarrow \mathbf{s}\ V}\,\textsf{ev\_s}$$

$$\frac{E_1 \hookrightarrow \mathbf{z} \qquad E_2 \hookrightarrow V}{(\mathbf{case}\ E_1\ \mathbf{of}\ \mathbf{z} \Rightarrow E_2 \mid \mathbf{s}\ x \Rightarrow E_3) \hookrightarrow V}\,\textsf{ev\_case\_z}$$

$$\frac{E_1 \hookrightarrow \mathbf{s}\ V_1' \qquad [V_1'/x]E_3 \hookrightarrow V}{(\mathbf{case}\ E_1\ \mathbf{of}\ \mathbf{z} \Rightarrow E_2 \mid \mathbf{s}\ x \Rightarrow E_3) \hookrightarrow V}\,\textsf{ev\_case\_s}$$

Recursion

$$\frac{[\mathbf{fix}\ x.\ E/x]E \hookrightarrow V}{\mathbf{fix}\ x.\ E \hookrightarrow V}\,\textsf{ev\_fix}$$

The structural and substitution properties for $\lambda_e^{\to\Box}$ extend to Mini-ML$_e^{\Box}$ 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. □

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 non-modal 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:

$$\mathcal{D} = \frac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ E_1 \hookrightarrow \mathbf{box}\ E_1' & [E_1'/u]E_2 \hookrightarrow V_2 \end{array}}{\mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 \hookrightarrow V_2}\,\textsf{ev\_let\_box}$$

$\cdot; \cdot \vdash^e E_1 : \Box B$ and	
$u{:}A; \cdot \vdash^e E_2 : A$	by inversion
$\cdot; \cdot \vdash^e \mathbf{box}\ E_1' : \Box B$	by ind. hyp.
$\cdot; \cdot \vdash^e E_1' : B$	by inversion
$\cdot; \cdot \vdash^e [E_1'/u]E_2 : A$	by substitution lemma
$\cdot; \cdot \vdash^e V_2 : A$	by ind. hyp.

$\square$

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

Suppose that $\cdot;\cdot \vdash^e E : \square A$ 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 Example: The Power Function in Explicit Form

We now define the power function in Mini-ML$_e^\square$ in such a way that it has type $\mathsf{nat} \to \square(\mathsf{nat} \to \mathsf{nat})$, assuming a closed term $\mathit{times}{:}\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ (definable in the Mini-ML fragment in the standard way).

$$\begin{array}{l} \mathit{power} \equiv \mathbf{fix}\ p{:}\mathsf{nat} \to \square(\mathsf{nat} \to \mathsf{nat}). \\ \qquad\qquad \lambda n{:}\mathsf{nat}.\ \mathbf{case}\ n \\ \qquad\qquad\quad \mathbf{of}\ \mathsf{z} \ \Rightarrow \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathsf{s}\ \mathsf{z}) \\ \qquad\qquad\quad \mid\ \mathsf{s}\ m \Rightarrow \mathbf{let}\ \ \mathbf{box}\ q = p\ m\ \mathbf{in} \\ \qquad\qquad\qquad\qquad\quad \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ (q\ x)) \end{array}$$

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

$$\begin{array}{rcl} \mathit{power}\ \mathsf{z} & \hookrightarrow & \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathsf{s}\ \mathsf{z}) \\ \mathit{power}\ (\mathsf{s}\ \mathsf{z}) & \hookrightarrow & \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ ((\lambda x{:}\mathsf{nat}.\ \mathsf{s}\ \mathsf{z})x)) \\ \mathit{power}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z})) & \hookrightarrow & \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x \\ & & \qquad ((\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ ((\lambda x{:}\mathsf{nat}.\ \mathsf{s}\ \mathsf{z})x))x)) \end{array}$$

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$_e^\square$ may be implemented by a translation into pure Mini-ML, by the mapping:

$$\begin{array}{rcll} \square A & \mapsto & 1 \to A & \\ \mathbf{box}\ E & \mapsto & \lambda x{:}1.\ E & \text{(where } x \text{ not free in } E\text{)} \\ \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 & \mapsto & (\lambda x{:}1 \to A.\ [x\ \langle\,\rangle/u]E_2)\ E_1 & \text{(where } x \text{ not in free } E_2\text{)} \end{array}$$

It may then appear that the modal fragment of Mini-ML$_e^\square$ is redundant. Note, however, that the type $1 \to A$ does not express any binding-time properties, while $\square A$ does. It is precisely this distinction which makes Mini-ML$_e^\Box$ interesting: The type checker will reject programs which 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 [KEH93]). For many applications this code would instead be source expressions or some intermediate language, thus allowing optimization after code substitution, as in partial evaluation [JGS93]. In our own experiments in run-time code generation [WLPD98, WLP98], following ideas in [LL94, LL96], 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 multiple-world interpretation of modal logic [Kri63]. A world corresponds to a stage of computation 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 $\lambda^{\to\Box}$ in [PW95] and [DP96]. 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 [NN92] 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

$$\Gamma_1; \Gamma_2; \ldots; \Gamma_n \overset{i}{\vdash} A$$

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 the hypotheses true in an *arbitrary* world reachable from Γ_1, and so on.

From the functional point of view, Γ_i binds variables available at stage i of the computation, where the proof term assigned to A may be executed at stage n. We refer to $\Gamma_1; \ldots; \Gamma_n$ 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.

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

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

Substitution Principle
If $\Psi; \Gamma \overset{i}{\vdash} M_1 : A$ *and* $\Psi; (\Gamma, x{:}A, \Gamma'); \Psi' \overset{i}{\vdash} M_2 : C$ *then* $\Psi; (\Gamma, \Gamma'); \Psi' \overset{i}{\vdash} [M_1/x]M_2 : C$.

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 jugdment

$$\Psi; \Gamma; \Gamma'; \Psi' \overset{i}{\vdash} M : C$$

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'.

$$\Psi; (\Gamma, \Gamma'); \Psi' \overset{i}{\vdash} M' : C$$

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

$$\Box(A \to B); A \overset{i}{\vdash} B$$

as

> *If* $\Box(A \to B)$ *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* $\Box(A \to B)$ *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

$$\Box(A \to B), A \overset{i}{\vdash} B$$

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

Next we add transitivity. Consider once again

$$\Psi; \Gamma; \Gamma'; \Psi' \overset{i}{\vdash} M : C.$$

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.

$$\Psi; \Gamma; \cdot; \Gamma'; \Psi' \overset{i}{\vdash} M' : C$$

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.

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

$$\frac{\Psi; \Gamma \overset{i}{\vdash} M : A \to B \qquad \Psi; \Gamma \overset{i}{\vdash} N : A}{\Psi; \Gamma \overset{i}{\vdash} M\,N : B} \to \mathsf{E}$$

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

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

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

$$\frac{\Psi; \Gamma \overset{i}{\vdash} M : \Box A}{\Psi; \Gamma; \Gamma_1; \ldots; \Gamma_n \overset{i}{\vdash} \mathbf{unbox}_n\, M : A} \Box\mathsf{E}$$

Note that Γ_1 is always accessible from Γ, so in K we only have $\mathbf{unbox}_1$. The worlds $\Gamma_2, \ldots, \Gamma_n$ 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.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Psi;\Gamma;\cdot \overset{i}{\vdash} M : A\end{array}}{\Psi;\Gamma \overset{i}{\vdash} \mathbf{box}\ M : \Box A}\ \Box\mathsf{I}}{\Psi;\Gamma;\Gamma_1;\ldots;\Gamma_n \overset{i}{\vdash} \mathbf{unbox}_n\ (\mathbf{box}\ M) : A}\ \Box\mathsf{E} \quad \Longrightarrow \quad \begin{array}{c}\mathcal{D}'\\ \Psi;\Gamma;\Gamma_1;\ldots;\Gamma_n \overset{i}{\vdash} M' : A\end{array}$$

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:

$$\begin{array}{c}\mathcal{D}\\ \Psi;\Gamma \overset{i}{\vdash} M : \Box A\end{array} \quad \Longrightarrow \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Psi;\Gamma \overset{i}{\vdash} M : \Box A\end{array}}{\Psi;\Gamma;\cdot \overset{i}{\vdash} \mathbf{unbox}_1\ M : A}\ \Box\mathsf{E}}{\Psi;\Gamma \overset{i}{\vdash} \mathbf{box}\ (\mathbf{unbox}_1\ M) : \Box A}\ \Box\mathsf{I}$$

We postpone a more 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 the index n in a term $\mathbf{unbox}_n M$ constructor 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	$::=$	$A_1 \to A_2 \mid \Box A$
Terms	M	$::=$	$x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid \mathbf{box}\ M \mid \mathbf{unbox}_n\ M$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}A$
Context Stacks	Ψ	$::=$	$\cdot \mid \Psi;\Gamma$

4.3 Natural Deduction Judgment

We summarize the rules defining the main judgment, $\Psi;\Gamma \overset{i}{\vdash} M : A$ as motivated and developed in Section 4.1.

$$\frac{x{:}A \text{ in } \Gamma}{\Psi;\Gamma \overset{i}{\vdash} x : A}\,\mathsf{var} \qquad\qquad \frac{\Psi;(\Gamma, x{:}A) \overset{i}{\vdash} M : B}{\Psi;\Gamma \overset{i}{\vdash} \lambda x{:}A.\ M : A \to B}\to \mathsf{I}$$

$$\frac{\Psi;\Gamma \overset{i}{\vdash} M : A \to B \qquad \Psi;\Gamma \overset{i}{\vdash} N : A}{\Psi;\Gamma \overset{i}{\vdash} M\,N : B}\to \mathsf{E}$$

$$\frac{\Psi;\Gamma;\cdot \overset{i}{\vdash} M : A}{\Psi;\Gamma \overset{i}{\vdash} \mathbf{box}\ M : \Box A}\,\Box\mathsf{I} \qquad\qquad \frac{\Psi;\Gamma \overset{i}{\vdash} M : \Box A}{\Psi;\Gamma;\Gamma_1;\ldots;\Gamma_n \overset{i}{\vdash} \mathbf{unbox}_n\, M : A}\,\Box\mathsf{E}$$

4.4 Properties of the Kripke-style λ-calculus

Structural transformations change the nature of the proof term by relabelling indices to the **unbox** constructor. Such a relabelling is also necessary to write out the rules β-reduction and η-expansion We define $\{n/m\}M$ inductively on the structure of M.

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

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

Lemma 9 (Modal Structural Transformation)
If $\Psi;\Gamma_0;\Delta_1;\cdots;\Delta_m \overset{i}{\vdash} M : A$ *then* $\Psi;\Gamma_0;\cdots;(\Gamma_n,\Delta_1);\cdots;\Delta_m \overset{i}{\vdash} \{n/m\}M : A$.

Proof: By induction over the structure of the given derivation $\mathcal{D}$ of $\Psi;\Gamma_0;\Delta_1;\ldots;\Delta_m$. In each case except for $\Box$E we immediately apply the induction hypothesis and reconstruct an appropriate derivation from the results. In the case of $\Box$E we distinguish two subcases.

Case:

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ \Psi;\Gamma_0;\Delta_1;\ldots;\Delta_{m-p} \overset{i}{\vdash} M_1 : \Box A\end{array}}{\Psi;\Gamma_0;\Delta_1;\ldots;\Delta_m \overset{i}{\vdash} \mathbf{unbox}_p M_1 : A}\,\Box\mathsf{E}$$

for $p < m$. Then

$\Psi;\Gamma_0;\ldots;(\Gamma_n,\Delta_1);\ldots;\Delta_{m-p} \overset{i}{\vdash} \{n/m-p\}M_1 : \Box A$	by ind. hyp.
$\Psi;\Gamma_0;\ldots;(\Gamma_n,\Delta_1);\ldots;\Delta_m \overset{i}{\vdash} \mathbf{unbox}_p\{n/m-p\}M_1 : A$	by rule $\Box$E
$\Psi;\Gamma_0;\ldots;(\Gamma_n,\Delta_1);\ldots;\Delta_m \overset{i}{\vdash} \{n/m\}\mathbf{unbox}_p M_1 : A$	by definition of $\{n/m\}$

Case:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi'; \Theta_{p-m} \overset{i}{\vdash} M_1 : \Box A\end{array}}{\Psi'; \Theta_{p-m}; \ldots; \Theta_0; \Delta_1; \ldots \Delta_m \overset{i}{\vdash} \mathbf{unbox}_p M_1 : A}\ \Box\mathsf{E}$$

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

$\Psi'; \Theta_{p-m}; \ldots; \Theta_0; \Gamma_1; \ldots; (\Gamma_n, \Delta_1); \ldots \Delta_m \overset{i}{\vdash} \mathbf{unbox}_{p+n-1} M_1 : A$ by rule $\Box\mathsf{E}$ applied to $\mathcal{D}_1$
$\Psi; \Gamma_0; \ldots; (\Gamma_n, \Delta_1); \ldots; \Delta_m \overset{i}{\vdash} \{n/m\}\mathbf{unbox}_p M_1 : A$ by definition of Ψ, Θ_0 and $\{n/m\}$.

□

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

Proof: By induction over the structure of the derivation of $\Psi; (\Gamma, x{:}A, \Gamma'); \Psi' \overset{i}{\vdash} M_2 : C$. □

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

$$\begin{array}{rcl}
(\lambda x{:}A.\ M)\ N & \overset{\beta}{\longmapsto} & [N/x]M \\
M : A_1 \to A_2 & \overset{\eta}{\longmapsto} & \lambda x{:}A_1.\ M\ x \\
\mathbf{unbox}_n\,(\mathbf{box}\,M) & \overset{\beta}{\longmapsto} & \{n/1\}M \\
M : \Box A & \overset{\eta}{\longmapsto} & \mathbf{box}\,(\mathbf{unbox}_1\,M)
\end{array}$$

Theorem 11 (Subject Reduction and Expansion)

1. *If $\Psi; \Gamma \overset{i}{\vdash} M : A$ and $M \overset{\beta}{\longmapsto} M'$ then $\Psi; \Gamma \overset{i}{\vdash} M' : A$.*
2. *If $\Psi; \Gamma \overset{i}{\vdash} M : A$ and $M : A \overset{\eta}{\longmapsto} M'$ then $\Psi; \Gamma \overset{i}{\vdash} M' : A$*

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.
□

4.5 Environments and Environment Stacks

In order to prove that the explicit and implicit 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 **box** u to explicit terms E.

Environments	ρ	$::=$	$\cdot \mid \rho, \mathbf{box}\ u = E$
Environment Stacks	R	$::=$	$\odot \mid R; \rho$

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.

$\Delta; \Gamma \vdash^{e} \rho : \Theta$	Environment ρ matches Θ in contexts Δ and Γ
$\Psi \models^{e} R : \Delta$	Environment stack R matches Δ in context stack Ψ

$$\frac{}{\Delta; \Gamma \vdash^{e} \cdot : \cdot}\ \textsf{tpv\_empty}$$

$$\frac{\Delta; \Gamma \vdash^{e} \rho : \Theta \qquad (\Delta, \Theta); \Gamma \vdash^{e} E : \Box A}{\Delta; \Gamma \vdash^{e} (\rho, \mathbf{box}\ u = E) : (\Theta, u{:}A)}\ \textsf{tpv\_bind}$$

$$\frac{}{\Psi \models^{e} \odot : \cdot}\ \textsf{tpr\_empty}$$

$$\frac{\Psi \models^{e} R : \Delta \qquad \Delta; \Gamma \vdash^{e} \rho : \Theta}{\Psi; \Gamma \models^{e} (R; \rho) : (\Delta, \Theta)}\ \textsf{tpr\_env}$$

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. $\Box$

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.

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

This judgment is defined by the following rules.

$$\frac{}{R;\rho \rhd x \mapsto x}\,\mathsf{tx\_ovar} \qquad \frac{R;\rho \rhd E \mapsto M}{R;\rho \rhd \lambda x{:}A.\ E \mapsto \lambda x{:}A.\ M}\,\mathsf{tx\_lam}$$

$$\frac{R;\rho \rhd E_1 \mapsto M_1 \qquad R;\rho \rhd E_2 \mapsto M_2}{R;\rho \rhd E_1\,E_2 \mapsto M_1\,M_2}\,\mathsf{tx\_app}$$

$$\frac{R;\rho;\cdot \rhd E \mapsto M}{R;\rho \rhd \mathbf{box}\ E \mapsto \mathbf{box}\ M}\,\mathsf{tx\_box}$$

$$\frac{R;\rho, \mathbf{box}\ u = E_1 \rhd E_2 \mapsto M}{R;\rho \rhd \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 \mapsto M}\,\mathsf{tx\_letbox}$$

$$\frac{R;\rho'_n \rhd E \mapsto M}{R;(\rho'_n, \mathbf{box}\ u = E, \rho''_n);\cdots;\rho_0 \rhd u \mapsto \mathbf{unbox}_n M}\,\mathsf{tx\_mvar}$$

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* $R;\rho \rhd E \mapsto M$.
2. *Whenever* $R;\rho \rhd E \mapsto M$ *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 $R;\rho \rhd E \mapsto M$, 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

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

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:

$$\mathcal{T} = \frac{}{R;\rho \rhd x \mapsto x}\,\mathsf{tx\_ovar}$$

$\Delta;\Gamma \vdash^e x : A$	by assumption
$x{:}A$ in Γ	by inversion
$\Psi;\Gamma \vdash^i x : A$	by rule var

Case:

$$\mathcal{T} = \dfrac{\begin{array}{c}\mathcal{T}_2\\ R;\rho \rhd E_2 \mapsto M_2\end{array}}{R;\rho \rhd \lambda x{:}A_1.\, E_2 \mapsto \lambda x{:}A_1.\, M_2}\,\mathsf{tx\_lam}$$

$\Delta;\Gamma \vdash^{e} \lambda x{:}A_1.\, E_2 : A$ by assumption
$\Delta;\Gamma, x{:}A_1 \vdash^{e} E_2 : A_2$ and $A = A_1 \to A_2$ by inversion
$\Psi;\Gamma, x{:}A_1 \models^{e} (R;\rho) : \Delta$ by assumption and weakening
$\Psi;\Gamma, x{:}A_1 \vdash^{i} M_2 : A_2$ by ind. hyp.
$\Psi;\Gamma \vdash \lambda x{:}A_1.\, M_2 : A_1 \to A_2$ by rule $\to\mathsf{I}$

Case: $\mathsf{tx\_app}$ is straightforward.

Case:

$$\mathcal{T} = \dfrac{\begin{array}{c}\mathcal{T}_1\\ R;\rho;\cdot \rhd E_1 \mapsto M_1\end{array}}{R;\rho \rhd \mathbf{box}\ E_1 \mapsto \mathbf{box}\ M_1}\,\mathsf{tx\_box}$$

$\Delta;\Gamma \vdash^{e} \mathbf{box}\ E_1 : A$ by assumption
$\Delta;\cdot \vdash^{e} E_1 : A_1$ and $A = \Box A_1$ by inversion
$\Psi;\Gamma \models^{e} (R;\rho) : \Delta$ by assumption
$\Delta;\cdot \vdash^{e} \cdot : \cdot$ by $\mathsf{tpv\_empty}$
$\Psi;\Gamma;\cdot \models^{e} (R;\rho;\cdot) : \Delta$ by $\mathsf{tpr\_env}$
$\Psi;\Gamma;\cdot \vdash^{i} M_1 : A_1$ by ind. hyp.
$\Psi;\Gamma \vdash^{i} \mathbf{box}\ M_1 : \Box A_1$ by $\Box\mathsf{I}$

Case:

$$\mathcal{T} = \dfrac{\begin{array}{c}\mathcal{T}_1\\ R;\rho, \mathbf{box}\ u = E_1 \rhd E_2 \mapsto M_2\end{array}}{R;\rho \rhd \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 \mapsto M_2}\,\mathsf{tx\_letbox}$$

$\Delta;\Gamma \vdash^{e} \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ E_2 : A$ by assumption
$\Delta;\Gamma \vdash^{e} E_1 : \Box A_1$ and
$(\Delta, u{:}A_1);\Gamma \vdash^{e} E_2 : A$ by inversion
$\Psi;\Gamma \models^{e} (R;\rho) : \Delta$ by assumption
$\Psi;\Gamma \models^{e} (R;\rho, \mathbf{box}\ u = E_1) : \Delta, u{:}A_1$ by Lemma 12
$\Psi;\Gamma \vdash^{i} M_2 : A$ by ind. hyp.

Case:

$$\mathcal{T} = \dfrac{\begin{array}{c}\mathcal{T}_1\\ R_n;\rho'_n \rhd E' \mapsto M'\end{array}}{R_n;(\rho'_n, \mathbf{box}\ u = E', \rho''_n);\cdots;\rho_0 \rhd u \mapsto \mathbf{unbox}_n M'}\,\mathsf{tx\_mvar}$$

$\Delta; \Gamma \vdash^{e} u : A$	by assumption
$u : A$ in Δ	by inversion
$\Psi; \Gamma_0 \models^{e} R_n; (\rho_n', \mathbf{box}\ u = e', \rho_n''); \cdots; \rho_0 : \Delta$ for $\Gamma_0 = \Gamma$ and $\rho_0 = \rho$	by assumption
$\Psi = \Psi_n; \Gamma_n; \ldots; \Gamma_1$,	
$\Delta = \Delta_{n+1}, \Theta_n, \ldots, \Theta_0$,	
$\Psi_n \models^{e} R_n : \Delta_{n+1}$, and	
$\Delta_{n+1}; \Gamma_n \vdash^{e} (\rho_n', \mathbf{box}\ u = E', \rho_n'') : \Theta_n$	by inversion
$\Delta_{n+1}; \Gamma_n \vdash^{e} (\rho_n', \mathbf{box}\ u = E') : \Theta_n', u{:}A'$ and $\Theta_n = \Theta_n', u{:}A', \Theta_n''$,	
$\Delta_{n+1}; \Gamma_n \vdash^{e} \rho_n' : \Theta_n'$ and	
$(\Delta_{n+1}, \Theta_n'); \Gamma_n \vdash^{e} E' : \Box A'$	by further inversion
$A' = A$	since $u : A$ in Δ is unique
$\Psi_n; \Gamma_n \models^{e} (R_n; \rho_n') : \Delta_{n+1}, \Theta_n'$	by rule tpr_env
$\Psi_n; \Gamma_n \vdash^{e} M' : \Box A$	by ind. hyp.
$\Psi_n; \Gamma_n; \ldots; \Gamma_0 \vdash^{e} \mathbf{unbox}_n M' : A$	by rule $\Box$E

$\Box$

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.

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

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

$$\frac{\Delta; \Gamma \vdash^{e} \rho : \Theta \qquad (\Delta, \Theta); \Gamma \vdash^{e} E : B}{\Delta; \Gamma \vdash^{e} \mathrm{Let}(\rho)(E) : B}\ \mathsf{tpi\_env}$$

The merge operation $R_1 \mid R_2$ 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.

$$\begin{array}{rcl} \odot \mid R_2 &=& R_2 \\ \mathrm{R}_1 \mid \odot &=& R_1 \\ (R_1; \rho_1) \mid (R_2; \rho_2) &=& (R_1 \mid R_2); (\rho_1, \rho_2) \end{array}$$

The translation is defined by the judgment

$M \mapsto R \rhd E \quad M$ compiles to term E under stack R

It is defined by the following rules.

$$\frac{}{x \mapsto \odot \rhd x}\,\textsf{tr\_var} \qquad \frac{M \mapsto R \rhd E}{\lambda x{:}A.\ M \mapsto R \rhd \lambda x{:}A.\ E}\,\textsf{tr\_lam}$$

$$\frac{M_1 \mapsto R_1 \rhd E_1 \qquad M_2 \mapsto R_2 \rhd E_2}{M_1\, M_2 \mapsto (R_1 \mid R_2) \rhd E_1\, E_2}\,\textsf{tr\_app}$$

$$\frac{M \mapsto \odot \rhd E}{\mathbf{box}\ M \mapsto \odot \rhd \mathbf{box}\ E}\,\textsf{tr\_box0} \qquad \frac{M \mapsto (R;\rho) \rhd E}{\mathbf{box}\ M \mapsto R \rhd \mathrm{Let}(\rho)(\mathbf{box}\ E)}\,\textsf{tr\_box1}$$

$$\frac{M \mapsto R \rhd E}{\mathbf{unbox}_0\ M \mapsto R \rhd \mathbf{let\ box}\ u = E\ \mathbf{in}\ u}\,\textsf{tr\_unbox0}$$

$$\frac{M \mapsto R \rhd E}{\mathbf{unbox}_{n+1}\ M \mapsto R; (\mathbf{box}\ u = E); \underbrace{\cdot\,; \ldots ; \cdot}_{n} \rhd u}\,\textsf{tr\_unbox1}$$

The $\textsf{tr\_app}$ 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 $M \mapsto R \rhd E$.*
2. *If $M \mapsto R \rhd E$ 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 **box** M, where exactly one of $\textsf{tr\_box0}$ and $\textsf{tr\_box1}$ apply.

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

Case:

$$\mathcal{T} = \frac{\begin{array}{c}\mathcal{T}_1\\ M_1 \mapsto \odot \rhd E_1\end{array}}{\mathbf{box}\ M_1 \mapsto \odot \rhd \mathbf{box}\ E_1}\,\textsf{tr\_box0}$$

$\Psi; \Gamma \vdash^i \mathbf{box}\ M_1 : A$	by assumption
$\Psi; \Gamma; \cdot \vdash^i M_1 : A_1$ and $A = \Box A_1$	by inversion
$\Psi; \Gamma \models^e \odot : \Delta_1$ and $\Delta_1; \cdot \vdash^e E_1 : A_1$ for some Δ_1	by ind. hyp.
$\Delta_1 = \cdot$	by inversion
$\Psi \models^e \odot : \cdot$	by $\textsf{tpr\_empty}$
$\cdot; \Gamma \vdash^e \mathbf{box}\ E_1 : \Box A_1$	by $\Box$I

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

Case:

$$\mathcal{T} = \cfrac{\begin{array}{c}\mathcal{T}_1\\ M_1 \mapsto (R;\rho) \rhd E_1\end{array}}{\mathbf{box}\ M_1 \mapsto R \rhd \mathrm{Let}(\rho)(\mathbf{box}\ E_1)}\ \mathsf{tr\_box1}$$

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

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

Case:

$$\mathcal{T} = \cfrac{\begin{array}{c}\mathcal{T}_1\\ M_1 \mapsto R_1 \rhd E_1\end{array}}{\mathbf{unbox}_0\, M_1 \mapsto R \rhd \mathbf{let\ box}\ u = E_1\ \mathbf{in}\ u}\ \mathsf{tr\_unbox0}$$

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

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

Case:

$$\mathcal{T} = \cfrac{\begin{array}{c}\mathcal{T}_1\\ M_1 \mapsto R \rhd E_1\end{array}}{\mathbf{unbox}_{n+1}\ M_1 \mapsto R; (\mathbf{box}\ u = E_1); \underbrace{\cdot; \ldots; \cdot}_{n} \rhd u}\ \mathsf{tr\_unbox1}$$

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

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

5 Modal Mini-ML: Implicit Formulation

We now define Mini-ML$^{\Box}$, 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$^{\Box}_{e}$ 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$^{\Box}$ if we prefer; it is now a matter of style rather than a property enforced in the language.

Another motivation for Mini-ML$^{\Box}$ 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$^{\Box}_{e}$ due to the different syntactic structuring required. Further, Mini-ML$^{\Box}$ 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$^{\Box}$ is a practical and theoretically well-founded basis for an extension of Standard ML. Indeed, experience with the two languages PML [WLP98] and Meta-ML [TS97, TBS98, MTBS99] 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 $\mathbf{unbox}_1$ to `unquote` (`,`). $\mathbf{unbox}_0$ corresponds to `eval`. More generally, $\mathbf{unbox}_n$ 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 [Smi84].

The operational semantics of the new system is given in terms of a type-preserving compilation to Mini-ML$^{\Box}_{e}$ 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.

$$
\begin{array}{lrcl}
\text{Types} & A & ::= & \mathsf{nat} \mid A_1 \to A_2 \mid A_1 \times A_2 \mid 1 \mid \Box A \\
\text{Terms} & M & ::= & x \mid \lambda x{:}A.\ M \mid M_1\, M_2 \\
 & & & \mid \mathbf{box}\ M \mid \mathbf{unbox}_n\ M \\
 & & & \mid \langle M_1, M_2 \rangle \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M \\
 & & & \mid \langle\,\rangle \\
 & & & \mid \mathbf{z} \mid \mathbf{s}\ M \mid (\mathbf{case}\ M_1\ \mathbf{of}\ \mathbf{z} \Rightarrow M_2 \mid \mathbf{s}\ x \Rightarrow M_3) \\
 & & & \mid \mathbf{fix}\ x{:}A.\ M
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Context Stacks} & \Psi & ::= & \cdot \mid \Psi; \Gamma
\end{array}
$$

5.2 Typing Rules

In this section we present typing rules for Mini-ML$^{\Box}$ using context stacks. The typing judgment has the form:

$\Psi; \Gamma \overset{i}{\vdash} M : A$ 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 $\mathbf{unbox}_n$. 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

$$\frac{x{:}A \text{ in } \Gamma}{\Psi; \Gamma \overset{i}{\vdash} x : A}\,\mathsf{tpi\_var} \qquad \frac{\Psi; (\Gamma, x{:}A) \overset{i}{\vdash} M : B}{\Psi; \Gamma \overset{i}{\vdash} \lambda x{:}A.\, M : A \to B}\,\mathsf{tpi\_lam}$$

$$\frac{\Psi; \Gamma \overset{i}{\vdash} M : A \to B \qquad \Psi; \Gamma \overset{i}{\vdash} N : A}{\Psi; \Gamma \overset{i}{\vdash} M\, N : B}\,\mathsf{tpi\_app}$$

Code

$$\frac{\Psi; \Gamma; \cdot \overset{i}{\vdash} M : A}{\Psi; \Gamma \overset{i}{\vdash} \mathbf{box}\ M : \Box A}\,\mathsf{tpi\_box} \qquad \frac{\Psi; \Gamma \overset{i}{\vdash} M : \Box A}{\Psi; \Gamma; \Gamma_1; \ldots; \Gamma_n \ \overset{i}{\vdash} \mathbf{unbox}_n\, M : A}\,\mathsf{tpi\_unbox}$$

Products

$$\frac{\Psi; \Gamma \overset{i}{\vdash} M_1 : A_1 \qquad \Psi; \Gamma \overset{i}{\vdash} M_2 : A_2}{\Psi; \Gamma \overset{i}{\vdash} \langle M_1, M_2 \rangle : A_1 \times A_2}\,\mathsf{tpi\_pair}$$

$$\frac{\Psi; \Gamma \overset{i}{\vdash} M : A_1 \times A_2}{\Psi; \Gamma \overset{i}{\vdash} \mathbf{fst}\ M : A_1}\,\mathsf{tpi\_fst} \qquad \frac{\Psi; \Gamma \overset{i}{\vdash} M : A_1 \times A_2}{\Psi; \Gamma \overset{i}{\vdash} \mathbf{snd}\ M : A_2}\,\mathsf{tpi\_snd}$$

$$\frac{}{\Psi; \Gamma \overset{i}{\vdash} \langle\,\rangle : 1}\,\mathsf{tpi\_unit}$$

Natural Numbers

$$\frac{}{\Psi; \Gamma \overset{i}{\vdash} \mathbf{z} : \mathsf{nat}}\,\mathsf{tpi\_z} \qquad \frac{\Psi; \Gamma \overset{i}{\vdash} M : \mathsf{nat}}{\Psi; \Gamma \overset{i}{\vdash} \mathbf{s}M : \mathsf{nat}}\,\mathsf{tpi\_s}$$

$$\frac{\Psi; \Gamma \overset{i}{\vdash} M_1 : \mathsf{nat} \quad \Psi; \Gamma \overset{i}{\vdash} M_2 : A \quad \Psi; (\Gamma, x{:}\mathsf{nat}) \overset{i}{\vdash} M_3 : A}{\Psi; \Gamma \overset{i}{\vdash} (\mathbf{case}\ M_1\ \mathbf{of}\ \mathbf{z} \Rightarrow M_2 \mid \mathbf{s}\ x \Rightarrow M_3) : A}\,\mathsf{tpi\_case}$$

Recursion

$$\frac{\Psi; (\Gamma, x{:}A) \overset{i}{\vdash} M : A}{\Psi; \Gamma \overset{i}{\vdash} \mathbf{fix}\ x{:}A.\ M : A}\ \mathsf{tpi\_fix}$$

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$^{\Box}$ with a different syntactic structure than in Mini-ML$_e^{\Box}$, though still with type $\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat})$.

$$\begin{array}{l} \mathit{power} \equiv \mathbf{fix}\ p{:}\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat}). \\ \quad \lambda n{:}\mathsf{nat}.\ \mathbf{case}\ n \\ \quad\quad \mathbf{of}\ \mathsf{z} \quad \Rightarrow \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathsf{s}\ \mathsf{z}) \\ \quad\quad \mid\ \mathsf{s}\ m \Rightarrow \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ (\mathbf{unbox}_1\ (p\ m)\ x)) \end{array}$$

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

$$\begin{array}{l} \mathit{lift}_{\mathsf{nat}} \equiv \mathbf{fix} f{:}\mathsf{nat} \to \Box\mathsf{nat}. \\ \quad \lambda x{:}\mathsf{nat}.\ \mathbf{case}\ x \\ \quad\quad \mathbf{of}\ \mathsf{z} \quad \Rightarrow \mathbf{box}\ \mathsf{z} \\ \quad\quad \mid\ \mathsf{s}\ x' \Rightarrow \mathbf{box}\ (\mathsf{s}\ (\mathbf{unbox}_1\ (f\ x'))) \end{array}$$

A similar term of type $A \to \Box A$ that returns a **box**'ed copy of its argument exists exactly when every $\to$ in A is enclosed by a $\Box$. This justifies the inclusion of the *lift* primitive for base types in two-level languages such as in [GJ91], and it seems natural to include such a primitive in a realistic extension of our language, as in [WLP98].

5.4 Compilation to Explicit Language

We do not define an operational semantics for Mini-ML$^{\Box}$ directly; instead we depend upon a translation to Mini-ML$_e^{\Box}$. 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 [JS86].

The operational semantics induced by the translation is different from some obvious ones defined directly on Mini-ML$^{\Box}$. In [MM94], 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 [PW95] 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$_e^{\Box}$. We also have constructed a direct operational semantics for Mini-ML$^{\Box}$ generalizing [Hat95] that does capture staging, but prefer the compilation because it makes operational properties more evident. In particular, proving staging theorems for Mini-ML$^{\Box}$ directly would be much harder than taking advantage of the type-preserving compilation and proving the properties for Mini-ML$_e^{\Box}$.

6 A Two-level Language

In this section we define Mini-ML$_2$, a two-level functional language very close to the one described in [NN92]. We then define a simple translation into Mini-ML$^\Box$ and prove that binding-time correctness in Mini-ML$_2$ is equivalent to modal correctness of the translation in Mini-ML$^\Box$.

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 which 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

$$\underline{\lambda x}{:}\underline{\mathsf{nat}}.\ \overline{\mathbf{case}}\ \underline{x}\ \overline{\mathbf{of}}\ \overline{\mathsf{z}} \Rightarrow \overline{\mathsf{z}} \mid \overline{\mathsf{s}}\ \overline{y} \Rightarrow \overline{\mathsf{s}}\ \overline{\mathsf{z}}$$

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 [NN92] in that we inject *all* run-time types into compile-time types, instead of just function types. This follows [GJ91], 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 [NN92] 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

$$\begin{array}{lrcl}
\text{Run-time Types} & \tau & ::= & \underline{\mathsf{nat}} \mid \tau_1 \underline{\to} \tau_2 \mid \tau_1 \underline{\times} \tau_2 \mid \underline{1} \\
\text{Compile-time Types} & \sigma & ::= & \overline{\mathsf{nat}} \mid \sigma_1 \overline{\to} \sigma_2 \mid \sigma_1 \overline{\times} \sigma_2 \mid \overline{1} \mid \tau
\end{array}$$

$$\begin{array}{lrcl}
\text{Terms} & e & ::= & \underline{x} \mid \underline{\lambda x}{:}\tau.\ e \mid e_1 \mathbin{\underline{@}} e_2 \\
 & & & \mid \underline{\mathbf{fix}}\ \underline{x}{:}\tau.\ e \\
 & & & \mid \underline{\langle}\, e_1, e_2 \underline{\rangle} \mid \underline{\mathbf{fst}}\ e \mid \underline{\mathbf{snd}}\ e \\
 & & & \mid \underline{\mathsf{z}} \mid \underline{\mathsf{s}}\ e \\
 & & & \mid (\underline{\mathbf{case}}\ e_1\ \underline{\mathbf{of}}\ \underline{\mathsf{z}} \Rightarrow e_2 \mid \underline{\mathsf{s}}\ \underline{x} \Rightarrow e_3) \\
 & & & \mid \underline{\langle\rangle} \\
 & & & \mid \overline{y} \mid \overline{\lambda y}{:}\sigma.\ e \mid e_1 \mathbin{\overline{@}} e_2 \\
 & & & \mid \overline{\mathbf{fix}}\ \overline{y}{:}\sigma.\ e \\
 & & & \mid \overline{\langle}\, e_1, e_2 \overline{\rangle} \mid \overline{\mathbf{fst}}\ e \mid \overline{\mathbf{snd}}\ e \\
 & & & \mid \overline{\langle\rangle} \\
 & & & \mid \overline{\mathsf{z}} \mid \overline{\mathsf{s}}\ e \\
 & & & \mid (\overline{\mathbf{case}}\ e_1\ \overline{\mathbf{of}}\ \overline{\mathsf{z}} \Rightarrow e_2 \mid \overline{\mathsf{s}}\ \overline{y} \Rightarrow e_3) \\
\text{Run-time Contexts} & \Gamma & ::= & \cdot \mid \Gamma, \underline{x}{:}\tau \\
\text{Compile-time Contexts} & \Delta & ::= & \cdot \mid \Delta, \overline{y}{:}\sigma
\end{array}$$

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

$$\begin{array}{l} \mathit{power} \equiv \overline{\mathbf{fix}}\ \overline{p} : \overline{\mathsf{nat}} \mathbin{\overline{\rightarrow}} (\underline{\mathsf{nat}} \mathbin{\underline{\rightarrow}} \underline{\mathsf{nat}}). \\ \qquad\qquad \overline{\lambda}\overline{n} : \overline{\mathsf{nat}}.\ \overline{\mathbf{case}}\ \overline{n} \\ \qquad\qquad\quad \overline{\mathbf{of}}\ \overline{\mathsf{z}} \quad \Rightarrow (\underline{\lambda x}{:}\underline{\mathsf{nat}}.\ \underline{\mathsf{s}}\ \underline{\mathsf{z}}) \\ \qquad\qquad\quad \mid\ \overline{\mathsf{s}}\ \overline{m} \Rightarrow (\underline{\lambda x}{:}\underline{\mathsf{nat}}.\ \underline{\mathit{times}} \mathbin{\underline{@}} \underline{x} \mathbin{\underline{@}} ((\overline{p} \mathbin{\overline{@}} \overline{m}) \mathbin{\underline{@}} \underline{x})) \end{array}$$

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$^{\Box}$ 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 $\mathbf{unbox}_1$ constructor. It is easy to see that in this example we obtain the *power* function in implicit form, exactly as in Section 5.3:

$$\begin{array}{l} \mathit{power} \equiv \mathbf{fix}\ p{:}\mathsf{nat} \rightarrow \Box(\mathsf{nat} \rightarrow \mathsf{nat}). \\ \qquad\qquad \lambda n{:}\mathsf{nat}.\ \mathbf{case}\ n \\ \qquad\qquad\quad \mathbf{of}\ \mathsf{z} \quad \Rightarrow \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathsf{s}\ \mathsf{z}) \\ \qquad\qquad\quad \mid\ \mathsf{s}\ m \Rightarrow \mathbf{box}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ (\mathbf{unbox}_1\ (p\ m)\ x)) \end{array}$$

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:

$\Delta; \Gamma \vdash^{r} e : \tau$	expression e has run-time type τ
$\Delta \vdash^{c} e : \sigma$	expression e has compile-time type σ

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

$$\frac{\underline{x}{:}\tau \text{ in } \Gamma}{\Delta; \Gamma \vdash^{r} \underline{x} : \tau}\,\mathsf{tpr\_var} \qquad\qquad \frac{\Delta; (\Gamma, \underline{x}{:}\tau_2) \vdash^{r} e : \tau}{\Delta; \Gamma \vdash^{r} \underline{\lambda x}{:}\tau_2.\ e : \tau_2 \mathbin{\underline{\rightarrow}} \tau}\,\mathsf{tpr\_lam}$$

$$\frac{\Delta; \Gamma \vdash^{r} e_1 : \tau_2 \mathbin{\underline{\rightarrow}} \tau \qquad \Delta; \Gamma \vdash^{r} e_2 : \tau_2}{\Delta; \Gamma \vdash^{r} e_1 \mathbin{\underline{@}} e_2 : \tau}\,\mathsf{tpr\_app}$$

Products

$$\frac{\Delta;\Gamma \vdash^{r} e_1 : \tau_1 \qquad \Delta;\Gamma \vdash^{r} e_2 : \tau_2}{\Delta;\Gamma \vdash^{r} \underline{\langle}\, e_1, e_2 \,\underline{\rangle} : \tau_1 \,\underline{\times}\, \tau_2}\,\textsf{tpr\_pair}$$

$$\frac{\Delta;\Gamma \vdash^{r} e : \tau_1 \,\underline{\times}\, \tau_2}{\Delta;\Gamma \vdash^{r} \underline{\mathbf{fst}}\; e : \tau_1}\,\textsf{tpr\_fst} \qquad \frac{\Delta;\Gamma \vdash^{r} e : \tau_1 \,\underline{\times}\, \tau_2}{\Delta;\Gamma \vdash^{r} \underline{\mathbf{snd}}\; e : \tau_2}\,\textsf{tpr\_snd}$$

$$\frac{}{\Delta;\Gamma \vdash^{r} \underline{\langle\,\rangle} : \underline{1}}\,\textsf{tpr\_unit}$$

Natural Numbers

$$\frac{}{\Delta;\Gamma \vdash^{r} \underline{\mathbf{z}} : \underline{\textsf{nat}}}\,\textsf{tpr\_z} \qquad \frac{\Delta;\Gamma \vdash^{r} e : \underline{\textsf{nat}}}{\Delta;\Gamma \vdash^{r} \underline{\mathbf{s}}\; e : \underline{\textsf{nat}}}\,\textsf{tpr\_s}$$

$$\frac{\Delta;\Gamma \vdash^{r} e_1 : \underline{\textsf{nat}} \quad \Delta;\Gamma \vdash^{r} e_2 : \tau \quad \Delta;(\Gamma, \underline{x} : \underline{\textsf{nat}}) \vdash^{r} e_3 : \tau}{\Delta;\Gamma \vdash^{r} (\underline{\mathbf{case}}\; e_1\; \underline{\mathbf{of}}\; \underline{\mathbf{z}} \Rightarrow e_2 \mid \underline{\mathbf{s}}\; \underline{x} \Rightarrow e_3) : \tau}\,\textsf{tpr\_case}$$

Recursion

$$\frac{\Delta;(\Gamma, \underline{x}{:}\tau) \vdash^{r} e : \tau}{\Delta;\Gamma \vdash^{r} \underline{\mathbf{fix}}\; \underline{x}{:}\tau.\, e : \tau}\,\textsf{tpr\_fix}$$

Phase Transitions

$$\frac{\Delta \vdash^{c} e : \tau}{\Delta;\Gamma \vdash^{r} e : \tau}\,\textsf{down} \qquad \frac{\Delta;\cdot \vdash^{r} e : \tau}{\Delta \vdash^{c} e : \tau}\,\textsf{up}$$

Functions

$$\frac{\overline{y}{:}\sigma \text{ in } \Delta}{\Delta \vdash^{c} \overline{y} : \sigma}\,\textsf{tpc\_var} \qquad \frac{\Delta, \overline{y}{:}\sigma_2 \vdash^{c} e : \sigma}{\Delta \vdash^{c} \overline{\lambda}\overline{y}{:}\sigma_2.\, e : \sigma_2 \,\overline{\Rightarrow}\, \sigma}\,\textsf{tpc\_lam}$$

$$\frac{\Delta \vdash^{c} e_1 : \sigma_2 \,\overline{\Rightarrow}\, \sigma \qquad \Delta \vdash^{c} e_2 : \sigma_2}{\Delta \vdash^{c} e_1 \,\overline{@}\, e_2 : \sigma}\,\textsf{tpc\_app}$$

Products

$$\frac{\Delta \vdash^c e_1 : \sigma_1 \qquad \Delta \vdash^c e_2 : \sigma_2}{\Delta \vdash^c \overline{\langle} e_1, e_2 \overline{\rangle} : \sigma_1 \mathbin{\overline{\times}} \sigma_2}\,\textsf{tpc\_pair}$$

$$\frac{\Delta \vdash^c e : \sigma_1 \mathbin{\overline{\times}} \sigma_2}{\Delta \vdash^c \overline{\mathbf{fst}}\ e : \sigma_1}\,\textsf{tpc\_fst} \qquad \frac{\Delta \vdash^c e : \sigma_1 \mathbin{\overline{\times}} \sigma_2}{\Delta \vdash^c \overline{\mathbf{snd}}\ e : \sigma_2}\,\textsf{tpc\_snd}$$

$$\frac{}{\Delta \vdash^c \overline{\langle\,\rangle} : \overline{1}}\,\textsf{tpc\_unit}$$

Natural Numbers

$$\frac{}{\Delta \vdash^c \overline{\mathbf{z}} : \overline{\textsf{nat}}}\,\textsf{tpc\_z} \qquad \frac{\Delta \vdash^c e : \overline{\textsf{nat}}}{\Delta \vdash^c \overline{\mathbf{s}}\ e : \overline{\textsf{nat}}}\,\textsf{tpc\_s}$$

$$\frac{\Delta \vdash^c e_1 : \overline{\textsf{nat}} \quad \Delta \vdash^c e_2 : \sigma \quad \Delta, \overline{y} : \overline{\textsf{nat}} \vdash^c e_3 : \sigma}{\Delta \vdash^c (\overline{\mathbf{case}}\ e_1\ \overline{\mathbf{of}}\ \overline{\mathbf{z}} \Rightarrow e_2 \mid \overline{\mathbf{s}}\ \overline{y} \Rightarrow e_3) : \sigma}\,\textsf{tpc\_case}$$

Recursion

$$\frac{\Delta, \overline{y}{:}\sigma \vdash^c e : \sigma}{\Delta \vdash^c \overline{\mathbf{fix}}\ \overline{y}{:}\sigma.\ e : \sigma}\,\textsf{tpc\_fix}$$

Note that we remove run-time assumptions at the down rule, while in [NN92] this is done later at the up rule. This 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$^\Box$ is now very simple. We translate both run-time and compile-time Mini-ML fragments directly, and insert $\Box$, **box** and $\mathbf{unbox}_1$ 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 $\overline{e}$ to match any term whose top constructor matches the phase annotation.

Type Translation

$$\begin{aligned} \|\underline{\textsf{nat}}\| &= \textsf{nat} \\ \|\tau_1 \mathbin{\underline{\rightarrow}} \tau_2\| &= \|\tau_1\| \rightarrow \|\tau_2\| \\ \|\tau_1 \mathbin{\underline{\times}} \tau_2\| &= \|\tau_1\| \times \|\tau_2\| \\ \|\underline{1}\| &= 1 \end{aligned} \qquad \begin{aligned} |\overline{\textsf{nat}}| &= \textsf{nat} \\ |\sigma_1 \mathbin{\overline{\rightarrow}} \sigma_2| &= |\sigma_1| \rightarrow |\sigma_2| \\ |\sigma_1 \mathbin{\overline{\times}} \sigma_2| &= |\sigma_1| \times |\sigma_2| \\ |\overline{1}| &= 1 \\ |\tau| &= \Box\|\tau\| \end{aligned}$$

Term Translation

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

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

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

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

Context Translation

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

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$^\Box$. Note that even though we use Δ and Γ to denote contexts, the implicit language Mini-ML$^\Box$ employs context stacks, where $\cdot; \Gamma_1; \ldots; \Gamma_n$ is abbreviated as $\Gamma_1; \ldots; \Gamma_n$.

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$^\Box$ and Mini-ML$_2$, 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: $\|\overline{e}\| = \mathbf{unbox}_1\ |\overline{e}|$, part 1a.

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

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

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

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

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

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

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

□

The translation and proof can be easily generalized from a two-level language to a B-level language [NN92] with an infinite linear ordering. In this case the image of the translation on well-typed terms is exactly the fragment Mini-ML$^{\Box}_{K}$, 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 [MM94], and which we discussed briefly in Section 2. Thus a corollary of the generalized theorem is that Mini-ML$^{\Box}_{K}$ is equivalent to an infinite linear B-level language, since the translation is then a bijection which preserves correctness of typing.

7 Examples

We now present some standard examples from partial evaluation to illustrate the expressiveness of our language Mini-ML$^{\Box}$. We use **let** $x = E_1$ **in** E_2 to introduce (non-polymorphic) 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:

$$\begin{array}{l}
\mathbf{fix}\ acker{:}\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}.\\
\quad \lambda m{:}\mathsf{nat}.\ \mathbf{case}\ m\\
\qquad \mathbf{of}\ \mathbf{z} \quad \Rightarrow \lambda n{:}\mathsf{nat}.\ \mathsf{s} n\\
\qquad \mid\ \mathsf{s}\ m' \Rightarrow \lambda n{:}\mathsf{nat}.\ \mathbf{case}\ n\\
\qquad\qquad\qquad \mathbf{of}\ \mathbf{z} \quad \Rightarrow acker\ m'\ (\mathsf{s}\ \mathbf{z})\\
\qquad\qquad\qquad \mid\ \mathsf{s}\ n' \Rightarrow acker\ m'\ (acker\ m\ n')
\end{array}$$

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

$$\begin{array}{l}
\mathbf{fix}\ acker{:}\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat}).\\
\quad \lambda m{:}\mathsf{nat}.\ \mathbf{case}\ m\\
\qquad \mathbf{of}\ \mathbf{z} \quad \Rightarrow \mathbf{box}\ (\lambda n{:}\mathsf{nat}.\ \mathsf{s} n)\\
\qquad \mid\ \mathsf{s}\ m' \Rightarrow \mathbf{box}\ (\lambda n{:}\mathsf{nat}.\ \mathbf{case}\ n\\
\qquad\qquad\qquad \mathbf{of}\ \mathbf{z} \quad \Rightarrow (\mathbf{unbox}_1\ (acker\ m'))\ (\mathsf{s}\ \mathbf{z})\\
\qquad\qquad\qquad \mid\ \mathsf{s}\ n' \Rightarrow (\mathbf{unbox}_1\ (acker\ m'))((\mathbf{unbox}_1(acker\ m))n'))
\end{array}$$

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.

$$\begin{array}{l}
\mathbf{fix}\ acker{:}\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat}).\\
\quad \lambda m{:}\mathsf{nat}.\ \mathbf{case}\ m\\
\qquad \mathbf{of}\ \mathbf{z} \quad \Rightarrow \mathbf{box}\ (\lambda n{:}\mathsf{nat}.\ \mathsf{s}\ n)\\
\qquad \mid\ \mathsf{s}\ m' \Rightarrow \mathbf{box}\ (\mathbf{fix}\ ackm.\ \lambda n{:}\mathsf{nat}.\\
\qquad\qquad\qquad \mathbf{case}\ n\\
\qquad\qquad\qquad\quad \mathbf{of}\ \mathbf{z} \quad \Rightarrow (\mathbf{unbox}_1\ (acker\ m'))\ (\mathsf{s}\ \mathbf{z})\\
\qquad\qquad\qquad\quad \mid\ \mathsf{s}\ n' \Rightarrow (\mathbf{unbox}_1\ (acker\ m'))\ (ackm\ n'))
\end{array}$$

This function will always terminate. The recursive applications appearing inside $\mathbf{unbox}_1$ constructors are evaluated when the first argument is given. The compilation of this function to Mini-ML$^{\Box}_{e}$ makes this more explicit:

$$\begin{array}{l}
\mathbf{fix}\ acker{:}\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat}).\\
\quad \lambda m{:}\mathsf{nat}.\ \mathbf{case}\ m\\
\qquad \mathbf{of}\ \mathbf{z} \quad \Rightarrow \mathbf{box}\ (\lambda n{:}\mathsf{nat}.\ \mathsf{s}\ n)\\
\qquad \mid\ \mathsf{s}\ m' \Rightarrow \mathbf{let\ box}\ f\ =\ acker\ m'\ \mathbf{in}\\
\qquad\qquad\qquad \mathbf{let\ box}\ g\ =\ acker\ m'\ \mathbf{in}\\
\qquad\qquad\qquad \mathbf{box}\ (\mathbf{fix}\ ackm.\ \lambda n{:}\mathsf{nat}.\\
\qquad\qquad\qquad\qquad \mathbf{case}\ n\ \mathbf{of}\ \mathbf{z} \quad \Rightarrow f\ (\mathsf{s}\ \mathbf{z})\\
\qquad\qquad\qquad\qquad\qquad\qquad \mid\ \mathsf{s}\ n' \Rightarrow g\ (ackm\ n'))
\end{array}$$

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 [GJ95] 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$^{\Box}$. We assume a data type vector in the example, along with a function $sub{:}\mathsf{nat} \to \mathsf{vector} \to \mathsf{nat}$ to access the elements of a vector.

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

$$
\begin{array}{l}
\mathbf{fix}\ ip{:}\mathsf{nat} \to \mathsf{vector} \to \mathsf{vector} \to \mathsf{nat}.\\
\lambda n{:}\mathsf{nat}.\ \mathbf{case}\ n\\
\quad \mathbf{of}\ \mathbf{z} \quad \Rightarrow \lambda v{:}\mathsf{vector}.\ \lambda w{:}\mathsf{vector}.\ \mathbf{z}\\
\quad \mid\ \mathsf{s}\ n' \Rightarrow \lambda v{:}\mathsf{vector}.\ \lambda w{:}\mathsf{vector}.\\
\qquad\qquad\qquad plus\ \ (times\ (sub\ n\ v)\ (sub\ n\ w))\\
\qquad\qquad\qquad\qquad (ip\ n'\ v\ w)
\end{array}
$$

We add in $\Box$, **box** and $\mathbf{unbox}_i$ to obtain a function with three computation stages which is shown in Figure 1. We assume a function $lift_{\mathsf{nat}}$ as defined earlier and a function $sub'{:}\mathsf{nat} \to \Box(\mathsf{vector} \to \mathsf{nat})$ 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.

$$
\begin{array}{l}
\mathbf{let}\ \ times'{:}\Box(\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat})) =\\
\qquad \mathbf{box}\ (\lambda m{:}\mathsf{nat}.\ \mathbf{case}\ m\\
\qquad\qquad \mathbf{of}\ \mathbf{z} \quad \Rightarrow \mathbf{box}\ (\lambda n{:}\mathsf{nat}.\ \mathbf{z})\\
\qquad\qquad \mid\ \mathsf{s}\ m' \Rightarrow \mathbf{box}\ (\lambda n{:}\mathsf{nat}.\ times\ n\ (\mathbf{unbox}_1\ (lift_{\mathsf{nat}}\ m))))\\
\mathbf{in\ let}\ \ iprod' = \mathbf{fix}\ ip{:}\mathsf{nat} \to \Box(\mathsf{vector} \to \Box(\mathsf{vector} \to \mathsf{nat})).\\
\qquad\qquad \lambda n{:}\mathsf{nat}.\ \mathbf{case}\ n\\
\qquad\qquad\quad \mathbf{of}\ \mathbf{z} \quad \Rightarrow \mathbf{box}\ (\lambda v{:}\mathsf{vector}.\ \mathbf{box}\ (\lambda w{:}\mathsf{vector}.\ \mathbf{z}))\\
\qquad\qquad\quad \mid\ \mathsf{s}\ n' \Rightarrow \mathbf{box}\ (\lambda v{:}\mathsf{vector}.\ \mathbf{box}\ (\lambda w{:}\mathsf{vector}.\\
\qquad\qquad\qquad\quad plus\ \ (\mathbf{unbox}_1\ \ (\mathbf{unbox}_1\ times'(\mathbf{unbox}_1\ (sub'\ n)\ v))\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathbf{unbox}_2\ (sub'\ n)\ w))\\
\qquad\qquad\qquad\qquad (\mathbf{unbox}_1\ (\mathbf{unbox}_1\ (ip\ n')\ v)\ w)))\\
\mathbf{in\ let}\ iprod3 : \mathsf{vector} \to \Box(\mathsf{vector} \to \mathsf{nat}) = \mathbf{unbox}_0(iprod'\ 3)\\
\mathbf{in\ let}\ iprod3a : \mathsf{vector} \to \mathsf{nat} = \mathbf{unbox}_0\ (iprod3\ [7, 0, 9])\\
\mathbf{in\ let}\ iprod3b : \mathsf{vector} \to \mathsf{nat} = \mathbf{unbox}_0\ (iprod3\ [7, 8, 0])\\
\mathbf{in}\ \ldots
\end{array}
$$

Figure 1: Staged code for inner product.

The last three lines show how to execute the result of a specialization using $\mathbf{unbox}_0$ (corresponding to eval in Lisp). Also, the occurrence of $\mathbf{unbox}_2$ 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 [GJ95].

Note the erasure of the $\mathbf{unbox}_i$ 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.

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 non-specializing 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
```

```
(*  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
```

Figure 2: Unstaged regular expression matcher

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)
               s
       in
           accStar k s
       end
```

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 [WLP98])

```
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 ‘ for **box** and ^ for $\mathbf{unbox}_1$. More generally, we suggest using ^n for $\mathbf{unbox}_n$.

```
(* 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)
                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
```

Figure 3: Modally staged regular expression matcher

We can now use our compilation to the explicit language Mini-ML$_e^\Box$ 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 unit -> A in this case, since the code constructor (‘) is only applied to values. We show this program only to demonstrate the staging described by the the modal annotated program. The program in Mini-ML$_e^\Box$ has the potential to be more efficient, since optimized code can be generated by a sophisticated implementation.

8 Related Work

Our modal $\lambda_e^{\rightarrow\Box}$-calculus is originally based on the modal λ-calculus presented by Bierman and de Paiva [BdP92] and used by Pfenning and Wong [PW95], who call it the "explicit system".

```
(* 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)
                 s
    in
        (fn k => fn s => acc2 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)
```

Figure 4: Pure SML staged regular expression matcher

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 [And92] for linear logic and by Girard [Gir93] for LU. The result is similar to the linear λ-calculus formulated by Wadler [Wad93].

The language Mini-ML$_e^\Box$ is constructed by combining $\lambda_e^{\rightarrow\Box}$ and Mini-ML [CDDK86]. The language Mini-ML$^\Box$ is based on the "implicit" modal λ-calculus presented in [PW95], which uses a stack of ordinary contexts rather than two contexts. Mini-ML$^\Box$ avoids the **pop** structural rule of [PW95], 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 [PW95]. Systems similar to the implicit modal λ-calculus of [PW95] have been proposed by Martini and Masini [MM94], who introduce a simple reduction semantics, and Bourghuis [Bor94], 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 [DP96], Goubault-Larrecq [GL96a, GL96b, GL96c, GL97] 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 [Mog89, Mog91] 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 current work (as yet unpublished) 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 [Kob97] and a related investigation by Benton, Bierman and de Paiva [BBdP98] who establish a connection between the computational λ-calculus and lax logic [FM97].

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 [WLP98] shows how this technique can be programmed in a language with modal types. See [BW93] for a description of a realistic partial evaluator for Standard ML and [JGS93] 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 [GJ91]. This is solved in Mini-ML$^\Box$ 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 [Jon91]. In other work [Dav96], 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.

Taha and Sheard [TS97] have directly constructed a language similar to Mini-ML$^\Box$ 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 [TBS98], 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 [MTBS99] and makes quite direct use of the results in the previous version of this paper [DP96] as well as [Dav96].

Over the last few years there has been a lot of interest in run-time code generation in high-level languages. Engler, Hsieh and Kaashoek [EHK96] describe an extension of the progamming language C called 'C (pronounced "tick C") which uses similar mechanisms to Mini-ML$^\Box$ 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 [CN96] 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 $\mathbf{unbox}_0$ construct in Mini-ML$^\Box$, while the annotations resulting from the binding-time analysis roughly correspond to the **box** and $\mathbf{unbox}_1$ constructs, although the restriction of Mini-ML$^\Box$ to closed code means that this correspondence is not exact. Leone and Lee [LL94, LL96] 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, Lee and Pfenning [WLP98] as an implementation technique for a language based on Mini-ML$_e^\Box$ 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$_e^\Box$ (including recursion, natural numbers, and products) and presented its natural operational semantics. We continued by defining an implicit language Mini-ML$^\Box$ 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$^\Box$ is given by a type-preserving compilation to Mini-ML$_e^\Box$. Further, Mini-ML$^\Box$ generalizes Nielson & Nielson's two-level functional language [NN92] 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$^\Box$ in the logic programming language Elf [Pfe91], 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 and Lee [WLPD98]. We hope that future design and implementation work will lead to a practical full-scale programming language with computation staging based on modal types.

Acknowledgements

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, and Philip Wickline for numerous fruitful discussions and suggestions.

References

[And92] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[BBdP98] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2), March 1998.

[BdP92] Gavin Bierman and Valeria de Paiva. Intuitionistic necessity revisited. In *Proceedings of the Logic at Work Conference*, Amsterdam, Holland, December 1992.

[Bor94] Tijn Borghuis. *Coming to Term with Modal Logic: On the Interpretation of Modalities in Typed λ-calculus.* PhD thesis, Eindhoven University of Technology, 1994.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, University of Copenhagen, Department of Computer Science, 1993. Available as Technical Report DIKU-report 93/22.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[CN96] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, July 1996.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 258–270, January 1996.

[EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, January 1996.

[FM97] M. Fairlough and M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, August 1997.

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[GJ91] Carsten Gomard and Neil Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GJ95] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S.D. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, pages 259–278. Springer-Verlag LNCS 982, September 1995.

[GL96a] Jean Goubault-Larrecq. On computational interpretations of the modal logic S4 - I. Cut elimination. Technical Report 1996-35, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Karlsruhe, Germany, 1996.

[GL96b] Jean Goubault-Larrecq. On computational interpretations of the modal logic S4 - II. The $\lambda\mathtt{ev}Q$-Calculus. Technical Report 1996-34, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Karlsruhe, Germany, 1996.

[GL96c] Jean Goubault-Larrecq. On computational interpretations of the modal logic S4 - III. termination, confluence, conservativity of $\lambda\mathtt{ev}Q$ and $\lambda\mathtt{ev}Q_H$. Technical Report 1996-33, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Karlsruhe, Germany, 1996.

[GL97] Jean Goubault-Larrecq. On computational interpretations of the modal logic S4 - IIIb. confluence, termination of the $\lambda\mathtt{ev}Q_H$-calculus. Technical Report 3164, INRIA, France, May 1997.

[Hat95] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In S.D. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics and Programs (PLILP'95)*. Springer-Verlag LNCS 982, September 1995.

[JGS93] Neil D. Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993.

[Jon91] Neil D. Jones. Efficient algebraic operations on programs. In T. Rus, editor, *AMAST Preliminary Proceedings, University of Iowa*, April 1991. A version appears as a chapter in [JGS93].

[JS86] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg Beach, Florida, January 1986.

[KEH93] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report TR 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[Kob97] Satoshi Kobayashi. Monad as modality. *Theoretical Computer Science*, 175:29–74, 1997.

[Kri63] Saul A. Kripke. Semantic analysis of modal logic. I: Normal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[LL94] Mark Leone and Peter Lee. Lightweight run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'94)*, Orlando, Florida, June 1994. An earlier version appears as Carnegie Mellon School of Computer Science Technical Report CMU-CS-93-225, November 1993.

[LL96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, Pennsylvania, May 1996.

[ML85a] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[ML85b] Per Martin-Löf. Truth of a proposition, evidence of a judgement, validity of a proof. Notes to a talk given at the workshop *Theory of Meaning*, Centro Fiorentino di Storia e Filosofia della Scienza, June 1985.

[MM94] Simone Martini and Andrea Masini. A computational interpretation of modal proofs. In H. Wansing, editor, *Proof theory of Modal Logics*. Kluwer, 1994. Workshop proceedings.

[Mog89] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989. IEEE Computer Society Press.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MTBS99] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *Proceedings of the European Symposium on Programming*, pages 193–207, Amsterdam, March 1999.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[Pal93] Jens Palsberg. Correctness of binding time analysis. *Journal of Functional Programming*, 3(3):347–363, July 1993.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

[PW95] Frank Pfenning and Hao-Chi Wong. On a modal λ-calculus for S4. In S. Brookes and M. Main, editors, *Proceedings of the Eleventh Conference on Mathematical Foundations of Programming Sematics*, New Orleans, Louisiana, March 1995. Available as *Electronic Notes in Theoretical Computer Science*, Volume 1, Elsevier.

[Smi84] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, January 1984. ACM Press.

[TBS98] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pages 918–929, July 1998.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 203–217, June 1997.

[Wad93] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science*, Gdansk, August 1993. Springer-Verlag LNCS 711. Invited Talk.

[WLP98] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and modal-ML. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 224–235, Montreal, Canada, June 1998. ACM Press.

[WLPD98] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), September 1998.