A Modal Analysis of Staged Computation

Rowan Davies[*] and Frank Pfenning[†]
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

rowan@cs.cmu.edu and fp@cs.cmu.edu

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 functional languages. Our main technical result is a conservative embedding of Nielson & Nielson's two-level functional language in our language Mini-ML$^\Box$, thus proving that binding-time correctness is equivalent to modal correctness on this fragment. In addition Mini-ML$^\Box$ can also express immediate evaluation and sharing of code across multiple stages, thus supporting run-time code generation as well as partial evaluation.

1 Introduction

Dividing a computation into separate stages is a common informal technique in the derivation of algorithms. For example, instead of matching a string against a regular expression we may first compile a regular expression into a finite automaton and then execute the automaton on a given string. Partial evaluation divides the computation into two stages based on the early availability of some function arguments. Binding-time analysis statically determines what part of the computation may be carried out in the first phase, and what part remains 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 *all* computation pertaining to the statically available arguments 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 (*e.g.* [GJ91, Hen91]) shows that this view is indeed possible and fruitful.

Up to now these type systems have been motivated *algorithmically*, that is, they are explicitly designed to support specialization. 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. 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.

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. Each world in the Kripke semantics of modal logic corresponds to a stage in the computation. A term of type $\Box A$ corresponds to code to be executed in a future stage of the computation. The modal restrictions imposed on terms of type $\Box A$ guarantee that a function of type $B \rightarrow \Box A$ can carry out *all* computation concerned with its first argument while generating the residual code of type A.

We begin by considering Mini-ML$^\Box_e$, a formulation of intuitionistic modal S4 in which the permissible operations on code and the staging of computation are represented very explicitly. The presentation is new, but draws on ideas in [BdP92, PW95]. It is augmented with a fixpoint operator, natural numbers, and pairs and endowed with a natural call-by-value operational semantics along the lines of Mini-ML [CDDK86].

Mini-ML$^\Box_e$ is somewhat awkward because it requires the broad syntactic structuring of the program to directly reflect staging. We thus consider a more implicit formulation of S4 directly 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 a 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

[*]This work was 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).

[†]This work was sponsored in part 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.

To appear in the 23rd Annual ACM Symposium on Principles of Programming Languages, January 1996, St. Petersburg Beach, Florida. An earlier version appears as Technical Report CMU-CS-95-145, Carnegie Mellon School of Computer Science.

correctness is not entirely trivial.

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 Modal Mini-ML: Explicit Formulation

This section presents Mini-ML$_e^\Box$, a language that combines some elements of Mini-ML [CDDK86] with a modal λ-calculus for intuitionistic S4 following ideas in [BdP92, PW95]. 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 within the given space constraints.

A common feature of many types of staged computation is the manipulation of code in various forms. Macro expanders and partial evaluators typically manipulate source expressions, runtime code generators typically manipulate object code or some form of intermediate code. Starting from a typed language such as Mini-ML we thus 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 representation of code. In this way our type system can support diverse applications. 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.

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 the natural deduction variant of the modal necessitation rule: We can infer that **box** $E : \Box A$ from $E : A$ if all hypotheses of the latter derivation are of the form $x : \Box B$. Note that this is quite different from Moggi's computational λ-calculus [Mog89] which only distinguishes values from computations and does not allow us to express stage separation. Moreover, the 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.

Technically we enforce the restriction by introducing two contexts into the typing judgement $\Delta; \Gamma \vdash^e E : A$. The outer context Δ contains variables that may be bound only to code during evaluation; Γ contains variables that may be bound to arbitrary values. Only variables in Δ are permitted to occur free inside **box** expressions. This presentation simplifies that of the modal λ-calculus $\lambda_e^{\rightarrow\Box}$ from [BdP92, PW95] by eliminating the need for simultaneous substitution while preserving subject reduction.

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

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

Note that the opposite coercion, $\lambda x{:}A.$ **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 a static argument which we refer to dynamically (i.e., inside the **box**).

2.1 Syntax

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

We use A, B for types, Γ, Δ for contexts, and x for variables assuming that any variable can be declared at most once in a context. Bound variables may be renamed tacitly. We omit leading $\cdot$'s from contexts. We write $[E'/x]E$ 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'.

2.2 Typing Rules

Our typing rules for the Mini-ML fragment of the explicit language are completely standard. The problem of typing the modal fragment is well understood; we present here a variant of known systems [BdP92, PW95] with two contexts as motivated above.

$$\Delta; \Gamma \vdash^e E : A \qquad \begin{array}{l} E \text{ has type } A \text{ in modal context} \\ \Delta \text{ and non-modal context } \Gamma. \end{array}$$

Our system has the property that a valid term has a unique type and typing derivation.

λ-calculus Fragment

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

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

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

Mini-ML Fragment

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

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

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

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

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

Modal Fragment

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

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

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

The elimination rule for $\Box$, tpe_let_box, is the only one that introduces variables into the modal context.

2.3 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 as to possible lower-level implementations.

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

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.

$$E \hookrightarrow V \quad \text{Expression } E \text{ evaluates to value } V.$$

λ-calculus Fragment

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

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

Mini-ML Fragment

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

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

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

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

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

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

Modal Fragment

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

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

As usual, the critical step in the proof of type preservation or subject reduction is a substitution lemma. Due to the modal contexts its form is slightly unusual, so we state it here explicitly. Other standard properties such as weakening are completely straightforward.

Lemma 1 (Substitution)

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

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

Proof: By standard, straight-forward inductions on the typing derivations for E_2. $\Box$

Theorem 2 (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 properties 1 and 2. We thus show only the ev_let_box case in the proof of property 3.

Case: We have

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{E_1 \hookrightarrow \mathbf{box}\ E_1'} \qquad \overset{\mathcal{D}_2}{[E_1'/x]E_2 \hookrightarrow V_2}}{\mathbf{let\ box}\ x = E_1\ \mathbf{in}\ E_2 \hookrightarrow V_2}\ \mathsf{ev\_let\_box}$$

and, by inversion on the derivation of $E : A$,

$$\dfrac{\overset{\mathcal{E}_1}{\cdot;\cdot \vdash^e E_1 : \Box A} \qquad \overset{\mathcal{E}_2}{x{:}A;\cdot \vdash^e E_2 : B}}{\cdot;\cdot \vdash^e \mathbf{let\ box}\ x = E_1\ \mathbf{in}\ E_2 : B}\ \mathsf{tpe\_let\_box}$$

Then we apply the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{E}_1$ to deduce that $\cdot;\cdot \vdash^e \mathbf{box}\ E_1' : \Box A$. Now, again by inversion, the derivation of this judgment must have the form

$$\dfrac{\overset{\mathcal{D}_1'}{\cdot;\cdot \vdash^e E_1' : A}}{\cdot;\cdot \vdash^e \mathbf{box}\ E_1' : \Box A}\ \mathsf{tpe\_box}$$

Then, from the substitution lemma applied to $\mathcal{D}_1'$ and $\mathcal{E}_2$ we deduce that $\cdot;\cdot \vdash^e [E_1'/x]E_2 : B$. We can then use $\mathcal{D}_2$ and the induction hypothesis to deduce that $\cdot;\cdot \vdash^e V_2 : B$, as required. $\Box$

We can now justify the claim that the type system of Mini-ML$_e^\Box$ captures the separation of a computation into stages. We follow the basic criteria for correctness of [Pal93] in which a modular proof of correctness for binding-time analyzes was presented.

Suppose that $\cdot;\cdot \vdash^e E : \Box A$ and $E \hookrightarrow V$. By 1 and 3 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 "static" since they will not appear in the residual code E'.

Further, the body of a **box** constructor can be considered "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 using the variable bound by a **let box** elimination construct to indicate where the delayed computation should be performed.

2.4 Example: The Power Function in Explicit Form

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

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

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

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

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

2.5 Implementation Issues

The operational semantics of Mini-ML$_e^\Box$ may be implemented by a translation into pure Mini-ML, by the mapping:

$$\begin{array}{rcl}
\Box A & \mapsto & \mathsf{unit} \to A\\
\mathbf{box}\ E & \mapsto & \lambda u{:}\mathsf{unit}.\ E\\
\mathbf{let\ box}\ x = E_1\ \mathbf{in}\ E_2 & \mapsto & (\lambda x'{:}\mathsf{unit} \to A.\ [x'()/x]E_2)\ E_1.
\end{array}$$

It may then appear that the modal fragment of Mini-ML$_e^\Box$ is redundant. Note, however, that the type $\mathsf{unit} \to A$ does not express any binding-time properties, while $\Box A$ does. It is precisely this distinction which makes Mini-ML$_e^\Box$ interesting: The type checker will reject programs 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.

The intended 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 $\mathsf{ev\_let\_box}$ 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. The deferred compilation approach described in [LL94] would provide a more sophisticated implementation, supporting fast run-time generation of optimized code.

3 Modal Mini-ML: Implicit Formulation

We now define Mini-ML$^\Box$, an "implicit" formulation of modal Mini-ML following the pure system $\lambda^{\to\Box}$ in [PW95]. The main advantage of this system over the explicit language is that altering the staging of a computation often only requires the insertion or deletion of modal constructors. In contrast, Mini-ML$_e^\Box$ requires that the structure of the program exactly mirror the staging, since the only way to refer to results from a previous stage is using variables that are bound to code outside the enclosing **box** constructor. 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 4) which would be much more difficult for Mini-ML$_e^\Box$. Further, Mini-ML$^\Box$ is very similar to the quasi-quoting and eval mechanisms in LISP, which appear to be 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 5.3), Mini-ML$^\Box$ would be a practical and theoretically well-founded basis for an extension of Standard ML.

The operational semantics of the new system is given in terms of a type-preserving compilation to Mini-ML$_e^\Box$ which resembles the proof of equivalence between $\lambda_e^{\to\Box}$ and $\lambda^{\to\Box}$ given in [PW95]. Besides the differences in the explicit system mentioned earlier, we add here a term constructor **pop**. This means that typing derivations for valid terms are unique and the compilation from implicit to explicit terms is deterministic, avoiding some unpleasant problems concerning coherence.

The intuition for this system comes from the multiple-world or Kripke semantics for modal logic [Kri63]. We think of a world as representing a stage of the computation. Computation is postponed to another stage by applying the **box** operator to a term M to generate code. Code may be used in the *current* stage with the **unbox** operator, or in any *future* stage by repeated application of the **pop** operator. These correspond to reflexivity and transitivity of the accessibility relation between worlds in the Kripke semantics, further motivating our choice of the particular modal logic S4. For some applications, such as the two-level λ-calculus, weaker modal logics such as K are sufficient, as described in Section 4.4.

It may be helpful to consider the modal fragment of the implicit language to be a statically typed analog to the quasiquote mechanism in Scheme. Then **box** corresponds to `quasiquote` (`) and **unbox** (**pop** $\cdot$) to `unquote` (,). **unbox** alone corresponds to `eval`, while **pop** alone corresponds to quoting an expression generated with `unquote`. 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].

3.1 Syntax

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

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

3.2 Typing Rules

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

$$\Psi; \Gamma \overset{i}{\vdash} M : A \qquad \text{Term } M \text{ has type } A \text{ in local context } \Gamma \text{ under stack } \Psi.$$

Intuitively, each element Δ of the context stack Ψ corresponds to a computation stage. The variables declared in Δ 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 M is unfrozen using **unbox**. 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 rules tpi_unbox and tpi_pop.

λ-calculus Fragment

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

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

Mini-ML Fragment

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

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

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

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

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

Modal Fragment

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

$$\frac{\Psi;\Delta \overset{i}{\vdash} M : \Box A}{\Psi;\Delta;\Gamma \overset{i}{\vdash} \mathbf{pop}\ M : \Box A}\,\mathsf{tpi\_pop}$$

3.3 Examples in Implicit Form

We now show how we can define the power function in Mini-ML$^\Box$ in a simpler form than in Mini-ML$_e^\Box$, though still with type $\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat})$. We use $\mathbf{unbox}_i\ M$ as syntactic sugar for $\mathbf{unbox}\ (\mathbf{pop}^i\ M)$.

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

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:

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

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, in a more realistic version of our language, we would also include it as a primitive.

3.4 Translation to Explicit Language

We do not define an operational semantics for Mini-ML$^{\Box}$ directly; instead we depend upon a translation to Mini-ML$^{\Box}_{e}$. This translation recursively extracts terms inside n nested **pop** 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.

The compilation from implicit to explicit terms is perhaps most easily described and understood if we restrict **pop** to occur only immediately underneath an **unbox** or another **pop**. On the pure fragment terms then follow the grammar

$$\begin{array}{lrcl} \text{Terms} & M & ::= & x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid \mathbf{box}\ M \mid \mathbf{unbox}\ P \\ \text{Pops} & P & ::= & M \mid \mathbf{pop}\ P \end{array}$$

The extension to the full language including recursion is tedious but straightforward. Any term can be transformed to one satisfying our restriction by replacing isolated occurrences of **pop** M by **box** (**unbox** (**pop** (**pop** M))). We also define environments and stacks which bind patterns of the form **box** x to explicit terms E.

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

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

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

The merge operation on environment stacks just appends the corresponding environments:

$$\begin{array}{c} \cdot \mid R = R \mid \cdot = R \\ (R_1; \rho_1) \mid (R_2; \rho_2) = (R_1 \mid R_2); (\rho_1, \rho_2) \end{array}$$

There are two primary judgements:

$$\begin{array}{ll} M \mapsto R \rhd E & M \text{ compiles to term } E \text{ under stack } R \\ P \mapsto^{x} R \rhd \rho & P \text{ compiles to environment } \rho \\ & \text{under stack } R \text{ binding } x \end{array}$$

On the fragment we are considering they are defined by the following rules:

$$\frac{P \mapsto^{y} R \rhd \rho}{\mathbf{pop}\ P \mapsto^{y} (R; \rho) \rhd \cdot}\ \mathsf{pr\_pop}$$

$$\frac{M \mapsto R \rhd E}{M \mapsto^{y} R \rhd \mathbf{box}\ y = E}\ \mathsf{pr\_tm}$$

$$\frac{}{x \mapsto \cdot \rhd x}\ \mathsf{tr\_var} \qquad \frac{M \mapsto R \rhd E}{\lambda x{:}A.\ M \mapsto R \rhd \lambda x{:}A.\ E}\ \mathsf{tr\_lam}$$

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

$$\frac{M \mapsto (R; \rho) \rhd E}{\mathbf{box}\ M \mapsto R \rhd \mathrm{Let}(\rho)(\mathbf{box}\ E)}\ \mathsf{tr\_box}$$

$$\frac{P \mapsto^{y} R \rhd \rho}{\mathbf{unbox}\ P \mapsto R \rhd \mathrm{Let}(\rho)(y)}\ \mathsf{tr\_unbox}^{y}$$

In the rule $\mathsf{tr\_unbox}^{y}$ the variable y must be new, that is, it may not occur free in the conclusion. The environment ρ will always either be empty (in which case y is bound in R) or a singleton of the form **box** $y = E$. Recall that in the former case, $\mathrm{Let}(\cdot)(y) = y$.

The correctness proof for this translation requires definitions of well-typed environments and environment stacks. The latter ties in the context stacks of the implicit system. We use Θ to range over contexts.

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

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

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

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

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

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

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

Theorem 3 (Correctness of Compilation)

1. *For any M there exist unique R and E such that $M \mapsto R \rhd E$.*

2. *For any P and y there exist unique R and ρ such that $P \mapsto^{y} R \rhd \rho$.*

3. *If $\Psi; \Gamma \vdash^{i} M : A$ and $M \mapsto R \rhd E$ then for some Δ we have $\Psi \models^{e} R : \Delta$ and $\Delta; \Gamma \vdash^{e} E : A$.*

4. *If $\Psi; \Gamma \vdash^{i} P : \Box A$ and $P \mapsto^{y} R \rhd \rho$ then for some Δ and Θ we have $\Psi \models^{e} R : \Delta$ and $\Delta; \Gamma \vdash^{e} \rho : \Theta$ with $y : A$ in Δ or Θ.*

Proof: Propositions 1 and 2 are trivial, since the translations are defined structurally on M with unique results (modulo renaming of bound variables, of course). Propositions 3 and 4 follow by mutual induction on the structure of the derivations of $M \mapsto R \rhd E$ and $M \mapsto^y R \rhd \rho$. 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 show only one critical case in the proof of property 3.
Case:

$$\mathcal{T} = \dfrac{\overset{\mathcal{T}_1}{M_1 \mapsto (R;\rho) \ \rhd E_1}}{\mathbf{box}\ M_1 \mapsto R \ \rhd \mathrm{Let}(\rho)(\mathbf{box}\ E_1)}\,\textsf{tr\_box}$$

By inversion we also have $A = \Box A_1$ and

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Psi;\Gamma;\cdot \vdash^i M_1 : A_1}}{\Psi;\Gamma \vdash^i \mathbf{box}\ M_1 : \Box A_1}\,\textsf{tpi\_box}$$

By induction hypothesis on $\mathcal{T}_1$ and $\mathcal{D}_1$ we know there are Δ_1 and derivations

$$\overset{\mathcal{V}_1}{\Psi;\Gamma \models^e (R;\rho) : \Delta_1} \quad \text{and} \quad \overset{\mathcal{E}_1}{\Delta_1;\cdot \vdash^e E_1 : A_1}$$

By inversion on $\mathcal{V}_1$ we find that $\Delta_1 = (\Delta_1', \Theta)$ and

$$\overset{\mathcal{V}_1'}{\Psi \models^e R : \Delta_1'} \quad \text{and} \quad \overset{\mathcal{F}_1'}{\Delta_1';\Gamma \vdash^e \rho : \Theta}$$

Using the derived rule for environment abstraction we have

$$\dfrac{\overset{\mathcal{F}_1'}{\Delta_1';\Gamma \vdash^e \rho : \Theta} \qquad \dfrac{\overset{\mathcal{E}_1}{\Delta_1', \Theta;\cdot \vdash^e E_1 : A_1}}{\Delta_1', \Theta;\Gamma \vdash^e \mathbf{box}\ E_1 : \Box A_1}\,\textsf{tpe\_box}}{\Delta_1';\Gamma \vdash^e \mathrm{Let}(\rho)(\mathbf{box}\ E_1) : \Box A_1}\,\textsf{tpi\_env}$$

At this point the desired conclusion follows with $\Delta = \Delta_1'$ from $\mathcal{V}_1'$ and this derivation. $\Box$

As an example of the compilation, it maps the definition of *power* from Section 3.3 to the one in Section 2.4. Note that the restructuring achieved by the compiler is similar to a staging transformation [JS86].

The operational semantics induced by the translation is very different from the obvious ones defined directly on Mini-ML$^\Box$. In [MM94] 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.

4 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$^\Box$.

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.

4.1 Syntax

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

$$\begin{array}{llll}
\text{Terms} \quad e & ::= & \underline{x} & \mid \underline{\lambda x}{:}\tau.\ e \mid e_1 \underline{@} e_2 \\
 & & & \mid \underline{\mathbf{fix}}\ \underline{x}{:}\tau.\ e \\
 & & & \mid \underline{\langle}\, e_1, e_2 \underline{\rangle} \mid \underline{\mathbf{fst}}\ e \mid \underline{\mathbf{snd}}\ e \\
 & & & \mid \underline{\mathbf{z}} \mid \underline{\mathbf{s}}\ e \\
 & & & \mid (\underline{\mathbf{case}}\ e_1\ \underline{\mathbf{of}}\ \underline{\mathbf{z}} \Rightarrow e_2 \mid \underline{\mathbf{s}}\ \underline{x} \Rightarrow e_3) \\
 & & \mid \overline{y} & \mid \overline{\lambda y}{:}\sigma.\ e \mid e_1 \overline{@} e_2 \\
 & & & \mid \overline{\mathbf{fix}}\ \overline{y}{:}\sigma.\ e \\
 & & & \mid \overline{\langle}\, e_1, e_2 \overline{\rangle} \mid \overline{\mathbf{fst}}\ e \mid \overline{\mathbf{snd}}\ e \\
 & & & \mid \overline{\mathbf{z}} \mid \overline{\mathbf{s}}\ e \\
 & & & \mid (\overline{\mathbf{case}}\ e_1\ \overline{\mathbf{of}}\ \overline{\mathbf{z}} \Rightarrow e_2 \mid \overline{\mathbf{s}}\ \overline{y} \Rightarrow e_3)
\end{array}$$

$$\begin{array}{llll}
\text{Run-time Contexts} & \Gamma & ::= & \cdot \mid \Gamma, \underline{x}{:}\tau \\
\text{Compile-time Contexts} & \Delta & ::= & \cdot \mid \Delta, \overline{y}{:}\sigma
\end{array}$$

4.2 Typing Rules

Run-time Typing

$$\dfrac{\underline{x}{:}\tau \text{ in } \Gamma}{\Delta;\Gamma \vdash^r \underline{x} : \tau}\,\textsf{tpr\_var}$$

$$\dfrac{\Delta;(\Gamma, \underline{x}{:}\tau_2) \vdash^r e : \tau}{\Delta;\Gamma \vdash^r \underline{\lambda x}{:}\tau_2.\ e : \tau_2 \underline{\to} \tau}\,\textsf{tpr\_lam}$$

$$\dfrac{\Delta;\Gamma \vdash^r e_1 : \tau_2 \underline{\to} \tau \qquad \Delta;\Gamma \vdash^r e_2 : \tau_2}{\Delta;\Gamma \vdash^r e_1 \underline{@} e_2 : \tau}\,\textsf{tpr\_app}$$

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

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

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

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

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

$$\frac{\Delta \vdash^c e : \tau}{\Delta; \Gamma \vdash^r e : \tau}\,\mathsf{down}$$

Compile-time Typing

$$\frac{\overline{y}{:}\sigma \text{ in } \Delta}{\Delta \vdash^c \overline{y} : \sigma}\,\mathsf{tpc\_var}$$

$$\frac{\Delta, \overline{y}{:}\sigma_2 \vdash^c e : \sigma}{\Delta \vdash^c \overline{\lambda y}{:}\sigma_2.\ e : \sigma_2 \overline{\rightarrow} \sigma}\,\mathsf{tpc\_lam}$$

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

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

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

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

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

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

$$\frac{\Delta; \cdot \vdash^r e : \tau}{\Delta \vdash^c e : \tau}\,\mathsf{up}$$

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.

4.3 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** , **unbox** and **pop** 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 between types, terms, and contexts. We write $\underline{e}$ and $\overline{e}$ to match any term whose top constructor matches the phase annotation.

Type Translation

$$\begin{array}{rcl@{\qquad}rcl}
\|\underline{\mathsf{nat}}\| & = & \mathsf{nat} & |\overline{\mathsf{nat}}| & = & \mathsf{nat} \\
\|\tau_1 \underline{\rightarrow} \tau_2\| & = & \|\tau_1\| \rightarrow \|\tau_2\| & |\sigma_1 \overline{\rightarrow} \sigma_2| & = & |\sigma_1| \rightarrow |\sigma_2| \\
\|\tau_1 \underline{\times} \tau_2\| & = & \|\tau_1\| \times \|\tau_2\| & |\sigma_1 \overline{\times} \sigma_2| & = & |\sigma_1| \times |\sigma_2| \\
 & & & |\tau| & = & \Box\|\tau\|
\end{array}$$

Term Translation

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

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

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

$$\|\overline{e}\| = \mathbf{unbox}\,(\mathbf{pop}\ |\overline{e}|) \qquad\qquad |\underline{e}| = \mathbf{box}\ \|\underline{e}\|$$

Context Translation

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

4.4 Equivalence of Binding Time Correctness and Modal Correctness

In this section we state our main theorem, which is that binding-time correctness is equivalent to modal correctness of the translation to Mini-ML$^\Box$. We write $\mathcal{D} :: (J)$ if $\mathcal{D}$ is a derivation of judgment J.

Theorem 4 (Conservative Embedding)

1. *If* $\|e\| = M$ *then:*
   (a) *if* $\mathcal{D}_r :: (\Delta; \Gamma \vdash^r e : \tau)$ *then*
   we have $\mathcal{D}_i :: (|\Delta|; \|\Gamma\| \vdash^i M : \|\tau\|)$*;*
   (b) *if* $\mathcal{D}_i :: (|\Delta|; \|\Gamma\| \vdash^i M : A)$ *then*
   we have $\mathcal{D}_r :: (|\Delta|; \|\Gamma\| \vdash^r e : \tau)$ *with* $\|\tau\| = A$*.*
2. *If* $|e| = M$ *then:*
   (a) *if* $\mathcal{D}_c :: (\Delta \vdash^c e : \sigma)$ *then*
   we have $\mathcal{D}_i :: (|\Delta| \vdash^i M : |\sigma|)$*;*
   (b) *if* $\mathcal{D}_i :: (|\Delta| \vdash^i M : A)$ *then*
   we have $\mathcal{D}_c :: (\Delta \vdash^c e : \sigma)$ *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 $\vdash^r$ judgements.

We only show the two cases involving both definitions, since all others are easy. Note that for variables we need to rely on the phase annotations.
Case: $\|\overline{e}\| = \mathbf{unbox}\,(\mathbf{pop}\ |\overline{e}|)$. To show part *1a* we note that by inversion we have

$$\mathcal{D}_r = \frac{\begin{array}{c}\mathcal{D}'_c \\ \Delta \vdash^c \overline{e} : \tau\end{array}}{\Delta; \Gamma \vdash^r \overline{e} : \tau}\,\mathsf{down}$$

Applying part *2a* of the induction hypothesis to $\mathcal{D}'_c$ yields $\mathcal{D}'_i$ from which we construct

$$\mathcal{D}_i = \dfrac{\dfrac{\begin{array}{c}\mathcal{D}'_i\\ |\Delta| \vdash^i |\overline{e}| : \Box\|\tau\|\end{array}}{|\Delta|; \|\Gamma\| \vdash^i \mathbf{pop}\ |\overline{e}| : \Box\|\tau\|}\,\mathsf{tpi\_pop}}{|\Delta|; \|\Gamma\| \vdash^i \mathbf{unbox}\ (\mathbf{pop}\ |\overline{e}|) : \|\tau\|}\,\mathsf{tpi\_unbox}$$

Now, to show part *1b* we note that we can reverse the roles of the inversion and proof construction above, and use part *2b* of the induction hypothesis.

Case: $|\underline{e}| = \mathbf{box}\ \|\underline{e}\|$ To show part *2a* we note that by inversion we have

$$\mathcal{D}_c = \dfrac{\begin{array}{c}\mathcal{D}'_r\\ \Delta; \cdot \vdash^r \underline{e} : \tau\end{array}}{\Delta \vdash^c \underline{e} : \tau}\,\mathsf{up}$$

Applying part *1a* of the induction hypothesis to $\mathcal{D}'_r$ yields $\mathcal{D}'_i$ from which we construct

$$\mathcal{D}_i = \dfrac{\begin{array}{c}\mathcal{D}'_i\\ |\Delta|; \cdot \vdash^i \|\underline{e}\| : \|\tau\|\end{array}}{|\Delta| \vdash^i \mathbf{box}\ \|\underline{e}\| : \Box\|\tau\|}\,\mathsf{tpi\_box}$$

Now, to show part *2b* we note that again we can reverse the roles of the inversion and proof construction and use part *1b* of the induction hypothesis.

□

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 **unbox** and **pop** are replaced by a combined constructor **unbox**$_1$. 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]. Thus a corollary of the generalized theorem is that Mini-ML$^{\Box}_{K}$ is equivalent to a B-level language, since the translation is then a typing-preserving bijection.

5 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 as syntactic sugar for $(\lambda x{:}A.\ E_2)\ E_1$.

5.1 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:

```
let ackermann =
      fix acker:nat → nat → nat.
        λm:nat. case m
          of z    ⇒ λn:nat. sn
          | s m′⇒ λn:nat. case n
                      of z   ⇒ acker m′ (s z)
                      | s n′⇒ (acker m′ (acker m n′))
in ...
```

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

```
let ackermann =
      fix acker:nat → □(nat → nat).
        λm:nat. case m
          of z    ⇒ box (λn:nat. sn)
          | s m′⇒ box (λn:nat. case n
                      of z   ⇒ (unbox_1 (acker m′)) (s z)
                      | s n′⇒ (unbox_1 (acker m′))
                                       ((unbox_1 (acker m))
                                                 n′))
in ...
```

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 **fix** as follows.

```
let ackermann =
      fix acker:nat → □(nat → nat).
        λm:nat. case m
          of z    ⇒ box (λn:nat. s n)
          | s m′⇒ box (fix ackm. λn:nat.
                      case n
                        of z   ⇒ (unbox_1 (acker m′)) (s z)
                         | s n′⇒ (unbox_1 (acker m′))
                                          (ackm n′))
in ...
```

This function will always terminate. The recursive applications appearing inside `unbox_1` constructors are evaluated when the first argument is given. The compilation of this function to Mini-ML$^{\Box}_{e}$ makes this more explicit:

```
let ackermann =
      fix acker:nat → □(nat → nat).
        λm:nat. case m
          of z    ⇒ box (λn:nat. s n)
          | s m′⇒ let box f  =  acker m′ in
                  let box g  =  acker m′ in
                  box (fix ackm. λn:nat.
                    case n
                      of z   ⇒ f (s z)
                        | s n′⇒ g (ackm n′))
in ...
```

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.

5.2 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*:nat → vector → nat to access the elements of a vector.

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

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

We add in $\Box$, **box** and $\mathbf{unbox}_i$ to get a function with three computation stages, which is shown in Figure 1. We assume a function $\mathit{lift}_{\mathsf{nat}}$ as defined earlier and a function $\mathit{sub}'{:}\mathsf{nat} \to \Box(\mathsf{vector} \to \mathsf{nat})$ which is a specializing version of *sub*, that perhaps pre-computes some pointer arithmetic based on the array index. We first define a staged version *times′* of *times* which avoids the multiplication in the specialization if the first argument is zero. This will speed up application of *iprod′* to its third argument, particularly in the case that the second argument is a sparse vector.

The last four lines show how to execute the result of a specialization using **unbox** without **pop** (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 a multi-level language.

Note the erasure of the $\mathbf{unbox}_i$ and **box** constructors in *iprod′* leaves *iprod*, except that we used a different version of multiplication. The operational semantics of the two programs is of course quite different.

5.3 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
```

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 `$` here to represent $\Box$)

```
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 $\mathbf{unbox}_1 \equiv \mathbf{unbox}\ (\mathbf{pop}\ \cdot)$. More generally, we suggest using $\hat{}\,n$ for $\mathbf{unbox}_n \equiv \mathbf{unbox}\ (\mathbf{pop}^n\ \cdot)$.

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

6 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 an explicit language Mini-ML$_e^\Box$ (including recursion, natural numbers, and pairs) and 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, perhaps with the addition of some syntactic sugar along the lines of Lisp's backquote and comma notation. 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, the main technical result of this paper.

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. Concrete instances such as partial evaluation, runtime code generation, or macro expansion will require some additional considerations for their effective use and efficient implementation.

For example, the two-level language we consider, Mini-ML$_2$, is directly based on the one in [NN92]. This has a stricter binding-time correctness criterion than used, for example, in [GJ91], even taking into account that the run-time part of the latter is dynamically typed. Essentially, this restriction may be traced to the fact that our underlying evaluation model applies only to closed terms, while [GJ91] requires manipulation of code with free variables. Thus, our system allows the inclusion of the **unbox** operator to evaluate closed code fragments, with no danger of encountering unbound variables.

Glück and Jørgensen [GJ95] have presented a multi-level binding-time analysis, along with practical motivations for multi-level partial evaluation, and also use the less strict binding-time correctness criterion. In other work [Dav95]

```
let  times′:□(nat → □(nat → nat)) =
       box (λm:nat. case m
                  of z   ⇒ box (λn:nat. z)
                  | s m′⇒ box (λn:nat. times n (unbox₁ (lift_nat m))))
in let  iprod′ = fix ip:nat → □(vector → □(vector → nat)).
                      λn:nat. case n
                        of z   ⇒ box (λv:vector. box (λw:vector. z))
                        | s n′⇒ box (λv:vector. box (λw:vector.
                                 plus (unbox₁ (unbox₁ times′(unbox₁ (sub′ n) v))
                                                 (unbox₂ (sub′ n) w))
                                      (unbox₁ (unbox₁ (ip n′) v) w)))
in let iprod3 : □(vector → □(vector → nat)) = iprod′ 3
in let iprod3a : □(vector → nat) = unbox iprod3 [7, 0, 9]
in let iprod3b : □(vector → nat) = unbox iprod3 [7, 8, 0]
in ...
```

Figure 1: Staged code for inner product.

one of the present authors has shown that the ◯ ("next") operator from non-branching temporal logic exactly models this looser correctness criterion. In future work we will consider how aspects of both type systems can be combined to allow both manipulation of code with free variables and type-safe evaluation of code in the same language.

Sheard and Nelson [SN95] have investigated a two-level extension of Standard ML based on [NN92], with the particular aim of statically typing program generators. They show that a form of dependent types allow a larger class of program generators to be typed. The use of two-level program generators is similar to using Mini-ML$^{\Box}$ to perform macro expansion, however Mini-ML$^{\Box}$ would also allow more general forms of computation staging. It is interesting to consider whether it would useful to add a form of dependent types to Mini-ML$^{\Box}$.

We have shown how some standard examples of specialization can be expressed in Mini-ML$^{\Box}$. More complicated examples would generally require polyvariant specialization, which could be provided to the programmer as a library function that memoized when generating code. Further, this approach could be extended to prevent code copying during specialization. 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.

Our language Mini-ML$^{\Box}$ requires the insertion of the **box**, **unbox** and **pop** operators into a functional program. These operators may be considered implicit coercions in a system of subtypes, where type inference corresponds to a form of generalized, polyvariant binding-time analysis. Principal types will only exist if we add restricted intersections, yielding a form of refinement types [FP91], but such a system would nonetheless raise serious coherence problems. Practical experience with larger examples will have to show how much inference along these lines is desirable and feasible.

We have omitted polymorphism in this paper, though ML-style polymorphism would only require the addition of a **let** construct. Variables thus bound would be restricted in the same way as λ-bound variables. Polymorphic code could be manipulated by also generalizing the types of variables bound by the **let box** elimination construct, with the familiar restrictions required in a call-by-value setting. Explicit polymorphism could also be added easily, following work on higher-order modal logics. Further, we expect our type system to interact very well with SML's module system. In fact, part of our original motivation was to provide the programmer with means to specify binding-time information in a signature and thus propagate it beyond module boundaries.

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. We have implemented a simple version of Mini-ML$^{\Box}$ in the logic programming language Elf [Pfe91]. To date we have only experimented with small examples, but we are planning a more realistic implementation to carry out larger experiments.

7 Acknowledgements

We gratefully acknowledge discussions with Lars Birkedal, Olivier Danvy, Joëlle Despeyroux, Andrzej Filinski, Robert Glück, Bob Harper, Fritz Henglein, Neil Jones, Jesper Jørgensen, Peter Lee, Mark Leone, Karoline Malmkjær, Greg Morrisett, Jens Palsberg, Morgan Price, Michael Schwartzbach, Morten Welinder, Hao-Chi Wong and David Wright regarding the subject of this paper.

```
(*  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

```
(* 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

```
(* 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

References

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

[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.

[Dav95] Rowan Davies. A temporal-logic approach to binding-time analysis. Research Series RS-95-51, BRICS, Department of Computer Science, University of Aarhus, October 1995.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[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*, pages 259–278. Springer-Verlag LNCS 982, September 1995.

[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*. Springer-Verlag LNCS 982, September 1995.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, pages 448–472. Springer-Verlag LNCS 523, 1991.

[JGS93] Neil D. Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 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.

[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. Deferred compilation: The automation of run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'94), Orlando*, June 1994. An earlier version appears as Carnegie Mellon School of Computer Science Technical Report CMU-CS-93-225, November 1993.

[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.

[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.

[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.

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

[SN95] Tim Sheard and Neal Nelson. Type safe abstractions using program generators. Technical Report OGI-TR-95-013, Oregon Graduate Institute of Science and Technology, Department of Computer Science, 1995.