A Modal Analysis of Staged Computation

Rowan Davies and Frank Pfenning

May 1995
CMU-CS-95-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as FOX Memorandum CMU-CS-FOX-95-02

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 functional languages. Our main technical result is a conservative embedding of Nielson & Nielson's two-level functional language in our language Mini-ML$^{\Box}$, which in addition to partial evaluation also supports multiple computation stages, sharing of code across multiple stages, and run-time code generation.

This research was sponsored by the Defense Advance Research Project Agency, 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.

Keywords: Modal Logic, Two Level Languages, Partial Evaluation, Run Time Code Generation

Contents

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 determines which part of the computation may be carried out in a first (static) phase, and which part remains to be done in a second (dynamic) 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 its types 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 partial evaluation. 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, sharing of code across multiple stages, and communication of binding-time information across module boundaries via types.

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 the current or a future stage of the computation. The modal restrictions imposed on a type of the form $\Box A$ guarantee that a function of type $B \to \Box A$ can carry out *all* computation concerned with its first argument while generating the residual code of type A.

The starting points for our investigation are the systems for the intuitionistic modal logic S4 in [BdP92, PW95] and the two-level λ-calculus in [NN92]. We augment the former with recursion to obtain Mini-ML$^\Box$ and then show that a two-level functional language may be fully and faithfully embedded in Mini-ML$^\Box$. This verifies that Mini-ML$^\Box$ is indeed a conservative extension of the two-level language of [NN92] and thus correctly expresses standard binding-time separation. Following [PW95], we also sketch a compilation from Mini-ML$^\Box$ to a related language Mini-ML$^\Box_e$ whose operational semantics embodies the separation of evaluation into multiple stages. The language Mini-ML$^\Box$ contains constructs similar to the Lisp backquote, comma and eval, allowing an existing program to be easily divided into stages, while the language Mini-ML$^\Box_e$ expresses the staging in a form that may be more directly executed.

2 Modal Mini-ML: An 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, $\lambda_e^{\rightarrow\Box}$ [BdP92, PW95]. The presentation of the modal constructs differs from $\lambda_e^{\rightarrow\Box}$ in that we have a **let** form for de-constructing boxed values, and use two contexts in the typing rules. This avoids the need for syntactic substitutions, but does not alter the essential properties of the system.

For the sake of simplicity, we make the language explicitly typed, since we do not treat type inference here. We have also chosen not to include polymorphism, because there are issues regarding the interaction between type variables and computation stages that would distract from the main point of this paper.

2.1 Syntax

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

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'. The addition of types $\Box A$ to Mini-ML introduces two new term constructs: **box** E for introduction and **let box** $x = E_1$ **in** E_2 for elimination.

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] inspired by zonal formulations of linear logic such as Girard's *LU* [Gir93]. In our formulation the typing judgment has two contexts. The first contains variables that may appear anywhere, since they represent code. The second contains variables that are only available in the current computation stage, including all ordinary Mini-ML variables. Thus our judgement

$$\Delta; \Gamma \vdash^e E : A$$

would correspond to $\Box\Delta, \Gamma \vdash E^* : A$ in $\lambda_e^{\rightarrow\Box}$, where E^* is an appropriate obvious translation of E. Note that our system has the property that a valid term has a unique typing derivation.

λ-calculus Fragment.

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

$$\frac{\Delta; \Gamma, x{:}A \vdash^{e} E : B}{\Delta; \Gamma \vdash^{e} \lambda x{:}A.\ E : A \rightarrow B}\ \textsf{tpe\_lam} \qquad \frac{\Delta; \Gamma \vdash^{e} E_1 : A \rightarrow B \qquad \Delta; \Gamma \vdash^{e} E_2 : A}{\Delta; \Gamma \vdash^{e} E_1\, E_2 : B}\ \textsf{tpe\_app}$$

Mini-ML Fragment.

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

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

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

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

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

Modal Fragment.

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

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

Note that the rule tpe_box does not allow variables bound in the second context to appear in the body of a **box** constructor, and only the rule tpe_let_box binds variables in the first context.

2.3 Operational Semantics

The Mini-ML fragment of our system has a standard operational semantics. For the modal part, we interpret **box** E as a value containing the frozen computation E which may be carried out in a future stage. We evaluate **let box** $x = E_1$ **in** E_2 as a substitution of the residual code generated by E_1 for x in E_2 and then evaluating E_2. The residual code for E_1 will then be evaluated during the evaluation of E_2 as necessary.

Note that if $E{:}A$ and $E \hookrightarrow V$ then $V{:}A$ and V is unique. Mini-ML has this property, which is easy to establish by induction over the structure of an evaluation. Also note that we have omitted types in terms from the rules below, since they are irrelevant here.

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

λ-calculus Fragment.

$$\frac{}{\lambda x.\, E \hookrightarrow \lambda x.\, E}\ \textsf{ev\_lam}$$

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

Mini-ML Fragment.

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

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

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

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

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

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

Modal Fragment.

$$\frac{}{\textbf{box } E \hookrightarrow \textbf{box } E}\,\textsf{ev\_box} \qquad \frac{E_1 \hookrightarrow \textbf{box } E_1' \quad [E_1'/x]E_2 \hookrightarrow V_2}{\textbf{let box } x = E_1 \textbf{ in } E_2 \hookrightarrow V_2}\,\textsf{ev\_let\_box}$$

Note that in the evaluation of well-typed terms, only terms inside a box constructor are ever substituted into another box constructor.

2.4 Example: The Power Function in Explicit Form

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

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

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

$$\begin{array}{rl} power\ \textsf{z} \hookrightarrow & \textbf{box } (\lambda x{:}\textsf{nat}.\ \textsf{s z}) \\ power\ (\textsf{s z}) \hookrightarrow & \textbf{box } (\lambda x{:}\textsf{nat}.\ times\ x\ ((\lambda x{:}\textsf{nat}.\ \textsf{s z})x)) \\ power\ (\textsf{s}\ (\textsf{s z})) \hookrightarrow & \textbf{box } (\lambda x{:}\textsf{nat}.\ times\ x\ ((\lambda x{:}\textsf{nat}.\ times\ x\ ((\lambda x{:}\textsf{nat}.\ \textsf{s z})x))x)) \end{array}$$

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

2.5 Implementation Issues

The operational semantics of Mini-ML$_e^\Box$ may be implemented by a translation into pure Mini-ML, mapping $\Box A$ to $\textsf{unit} \to A$; **box** E to $\lambda u{:}\textsf{unit}.\ E$; and **let box** $x = E_1$ **in** E_2 to $(\lambda x'{:}\textsf{unit} \to A.\ [x'()/x]E_2)E_1$. It may then appear that the modal fragment of Mini-ML$_e^\Box$ is redundant. Note, however, that the type $\textsf{unit} \to A$ 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.

Another implementation method would be to interpret $\Box A$ as a data-type representing code that calculates a value of type A. This code could be either machine code, source code, or some intermediate language. This would allow optimization after specialization, and could also support an operation to output code as a separate program. 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]). The deferred compilation approach in [LL94] would provide a more sophisticated implementation, supporting fast run-time generation of optimized code.

3 Modal Mini-ML: An Implicit Formulation

We now define an implicit version Mini-ML$^\Box$ of the explicit Mini-ML$^\Box_e$, following [PW95] where an implicit system $\lambda^{\rightarrow\Box}$ was defined. This system is more reasonable as a programming language, since we do not have to explicitly stage computation as required with **let box** $x = E_1$ **in** E_2. The operational semantics of the new system is given in terms of a type-preserving compilation to the explicit system. Our development differs from [PW95] in that we introduce 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.

3.1 Syntax

$$
\begin{array}{llcrl}
\text{Types} & A & ::= & \mathsf{nat} & \mid A_1 \rightarrow A_2 \mid A_1 \times A_2 \mid \Box A \\
\text{Terms} & M & ::= & x & \mid \lambda x{:}A.\ M \mid M_1\, M_2 \\
 & & & & \mid \mathbf{fix}\ x{:}A.\ M \mid \langle M_1, M_2\rangle \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M \\
 & & & & \mid \mathbf{z} \mid \mathbf{s}\ M \mid (\mathbf{case}\ M_1\ \mathbf{of}\ \mathbf{z} \Rightarrow M_2 \mid \mathbf{s}\ x \Rightarrow M_3) \\
 & & & & \mid \mathbf{box}\ M \mid \mathbf{unbox}\ M \mid \mathbf{pop}\ M \\
\text{Contexts} & \Gamma & ::= & \cdot & \mid \Gamma, x{:}A \\
\text{Context Stacks} & \Psi & ::= & \cdot & \mid \Psi; \Gamma
\end{array}
$$

All the categories, except *context stacks* are standard. The importance of context stacks will be apparent when we present the typing rules.

3.2 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.

The context stack enables the distinguished use of variables depending on their relative position with respect to the **box** operators that enclose the term being typed. 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 we enter a new evaluation stage, since M will be frozen during evaluation. 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 ("unboxed"). Thus, variables may only be looked up in the current (innermost) context (rule tpi_var) which is initialized as empty when we enter the scope 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.

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

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

Mini-ML Fragment.

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

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

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

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

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

Modal Fragment.

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

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

Note that it may be useful to consider the modal fragment of the implicit language to be a statically typed analogue to the quoting mechanism in Lisp. Then **box** corresponds to backquote and **unbox** (**pop** $\cdot$) to comma. **unbox** alone corresponds to eval, while **pop** alone corresponds to quoting an expression generated with comma. Note however that our implementation via a compilation to Mini-ML$_e^{\Box}$ is quite different from Lisp quoting.

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 $\mathsf{nat} \rightarrow \Box(\mathsf{nat} \rightarrow \mathsf{nat})$. We use $\mathbf{unbox}_i\ M$ as syntactic sugar for $\mathbf{unbox}\ (\mathbf{pop}^i\ M)$.

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

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

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

A similar term of type $A \rightarrow \Box A$ that returns a **box** 'ed copy of its argument exists exactly when A is observable, *i.e.*, contains no $\rightarrow$. This justifies the inclusion of the *lift* primitive in two-level languages such as in [GJ91], and in fact in a more realistic version of our language it could also be included 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$_e^{\Box}$. This translation recursively extracts terms inside a **pop** constructor and binds the result of their evaluation to new variables, bound with a **let box** outside the enclosing **box** constructor. Variables thus bound occur exactly once.

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

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

The extension to the full language including recursion is tedious but trivial. Any term can be transformed to one satisfying our restriction by replacing isolated occurrences of $\mathbf{pop}\ M$ by $\mathbf{box}\ (\mathbf{unbox}\ (\mathbf{pop}\ (\mathbf{pop}\ M)))$.

The compilation below keeps track of the context in which the term to be translated should be placed (the k argument). This is necessary so that when we encounter an **pop** operator we can find the matching **box** operator and insert a **let box** binding in the resulting explicit term. We use the notation $k = \Lambda h.\ E$ for a context k with hole h. Filling the hole is written as an application $k(E')$. This must be implemented as syntactic replacement since k is intended to capture variables free in E'. First, the translation on terms, $[\![M]\!]\,k$.

$$\begin{array}{lcl}
[\![x]\!]\,k & = & k(x) \\
[\![M_1\,M_2]\!]\,k & = & [\![M_1]\!](\Lambda h_1.\ [\![M_2]\!](\Lambda h_2.\ k(h_1\,h_2))) \\
[\![\lambda x.\ M]\!]\,k & = & [\![M]\!](\Lambda h.\ k(\lambda x.\ h)) \\
[\![\mathbf{box}\ M]\!]\,k & = & [\![M]\!](\Lambda h.\ k(\mathbf{box}\ h)) \\
[\![\mathbf{unbox}\ P]\!]\,k & = & [P]\,k\,(\Lambda h.\ h)
\end{array}$$

Nested **pop** operators are translated by traversing the current context k from the inside out until a **box** operator is found. This cancels one **pop** operator and continues the translation. After all **pop** operators have been removed (possibly none), we introduce a **let box** and continue the translation. The b argument accumulates the body of the **let box** which will eventually be introduced.

$$\begin{array}{lcl}
[\mathbf{pop}\ P]\,(\Lambda h.\ k(E_1\,h))\,b & = & [\mathbf{pop}\ P]\,k\,(\Lambda h.\ E_1\,b(h)) \\
[\mathbf{pop}\ P]\,(\Lambda h.\ k(h\,E_2))\,b & = & [\mathbf{pop}\ P]\,k\,(\Lambda h.\ b(h)\,E_2) \\
[\mathbf{pop}\ P]\,(\Lambda h.\ k(\lambda x.\ h))\,b & = & [\mathbf{pop}\ P]\,k\,(\Lambda h.\ \lambda x.\ b(h)) \\
[\mathbf{pop}\ P]\,(\Lambda h.\ k(\mathbf{box}\ h))\,b & = & [P]\,k\,(\Lambda h.\ \mathbf{box}\ b(h)) \\
[\mathbf{pop}\ P]\,(\Lambda h.\ k(\mathbf{let\ box}\ x = h\ \mathbf{in}\ E_2))\,b & = & \\
\multicolumn{3}{r}{[\mathbf{pop}\ P]\,k\,(\Lambda h.\ \mathbf{let\ box}\ x = b(h)\ \mathbf{in}\ E_2)} \\
[\mathbf{pop}\ P]\,(\Lambda h.\ k(\mathbf{let\ box}\ x = E_1\ \mathbf{in}\ h))\,b & = & \\
\multicolumn{3}{r}{[\mathbf{pop}\ P]\,k\,(\Lambda h.\ \mathbf{let\ box}\ x = E_1\ \mathbf{in}\ b(h))}
\end{array}$$

$$[M]\,k\,b = [\![M]\!]\,(\Lambda h.\ k(\mathbf{let\ box}\ y = h\ \mathbf{in}\ b(y))) \quad \text{where } y \text{ is new}$$

Since h must occur exactly once in $\Lambda h.\ E$, the cases for $[\mathbf{pop}\ P]\,k\,b$ leave out only $\Lambda h.\ h$. If the original term is well-typed this case can never arise. An important invariant of $[P]\,k\,b$ is that $\Lambda h.\ k(b(h))$ remains the same in every recursive call. At present we have not formally proven that the translation above maps well-typed explicit terms to well-typed implicit terms. A related, slightly more complicated translation has been proven correct in [PW95].

As an example of this translation, it maps the above definition of *power* to the previous explicit one.

It is important to note that the operational semantics induced by the translation is very different from the natural one defined directly on Mini-ML$^{\Box}$. In [MM94] a simple reduction semantics for a system similar to our implicit system is introduced which does not reflect binding time separation in any way. It is instead used to prove a Church-Rosser theorem and strong normalization for a pure modal λ-calculus.

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, which involves a small change in the "up" and "down" rules. All other differences to [NN92] are due to minor differences between their underlying language and Mini-ML.

4.1 Syntax

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

4.2 Typing Rules

Run-time Typing

$$\frac{\underline{x}{:}\tau \text{ in } \Gamma}{\Delta; \Gamma \vdash^{r} \underline{x} : \tau}\ \mathsf{tpr\_var}$$

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

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

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

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

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

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

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

Compile-time Typing

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

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

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

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

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

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

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

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

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 Modal Mini-ML

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 and terms. We write $\underline{e}$ and $\overline{e}$ to match any term whose top constructor matches the phase annotation.

Run-time Types

$$\begin{array}{rcl} \|\underline{\mathsf{nat}}\| & = & \mathsf{nat} \\ \|\tau_1 \underline{\rightarrow} \tau_2\| & = & \|\tau_1\| \rightarrow \|\tau_2\| \\ \|\tau_1 \underline{\times} \tau_2\| & = & \|\tau_1\| \times \|\tau_2\| \end{array}$$

Compile-time Types

$$
\begin{aligned}
|\overline{\mathsf{nat}}| &= \mathsf{nat} \\
|\sigma_1 \overline{\rightarrow} \sigma_2| &= |\sigma_1| \rightarrow |\sigma_2| \\
|\sigma_1 \overline{\times} \sigma_2| &= |\sigma_1| \times |\sigma_2| \\
|\tau| &= \Box \|\tau\|
\end{aligned}
$$

Run-time Terms

$$
\begin{aligned}
\|\underline{x}\| &= x \\
\|\underline{\lambda x}{:}\tau.\ e\| &= \lambda x{:}\|\tau\|.\ \|e\| \\
\|e_1 \underline{@} e_2\| &= \|e_1\|\ \|e_2\| \\
\|\underline{\mathbf{fix}}\ \underline{x}{:}\tau.\ e\| &= \mathbf{fix}\ x{:}\|\tau\|.\ \|e\| \\
\|\underline{\langle}\ e_1, e_2\ \underline{\rangle}\| &= \langle \|e_1\|, \|e_2\| \rangle \\
\|\underline{\mathbf{fst}}\ e\| &= \mathbf{fst}\ \|e\| \\
\|\underline{\mathbf{snd}}\ e\| &= \mathbf{snd}\ \|e\| \\
\|\underline{\mathbf{z}}\| &= \mathbf{z} \\
\|\underline{\mathbf{s}}\ e\| &= \mathbf{s}\ \|e\| \\
\|\underline{\mathbf{case}}\ e_1\ \underline{\mathbf{of}}\ \underline{\mathbf{z}} \Rightarrow e_2 \mid \underline{\mathbf{s}}\ \underline{x} \Rightarrow e_3\| &= \mathbf{case}\ \|e_1\|\ \mathbf{of}\ \mathbf{z} \Rightarrow \|e_2\| \mid \mathbf{s}\ x \Rightarrow \|e_3\| \\
\|\overline{e}\| &= \mathbf{unbox}\ (\mathbf{pop}\ |\overline{e}|)
\end{aligned}
$$

Compile-time Terms

$$
\begin{aligned}
|\overline{y}| &= y \\
|\overline{\lambda}\overline{y}{:}\tau.\ e| &= \lambda y{:}|\tau|.\ |e| \\
|e_1 \overline{@} e_2| &= |e_1|\ |e_2| \\
|\overline{\mathbf{fix}}\ \overline{y}{:}\tau.\ e| &= \mathbf{fix}\ y{:}|\tau|.\ |e| \\
|\overline{\langle}\ e_1, e_2\ \overline{\rangle}| &= \langle |e_1|, |e_2| \rangle \\
|\overline{\mathbf{fst}}\ e| &= \mathbf{fst}\ |e| \\
|\overline{\mathbf{snd}}\ e| &= \mathbf{snd}\ |e| \\
|\overline{\mathbf{z}}| &= \mathbf{z} \\
|\overline{\mathbf{s}}\ e| &= \mathbf{s}\ |e| \\
|\overline{\mathbf{case}}\ e_1\ \overline{\mathbf{of}}\ \overline{\mathbf{z}} \Rightarrow e_2 \mid \overline{\mathbf{s}}\ \overline{y} \Rightarrow e_3| &= \mathbf{case}\ |e_1|\ \mathbf{of}\ \mathbf{z} \Rightarrow |e_2| \mid \mathbf{s}\ y \Rightarrow |e_3| \\
|\underline{e}| &= \mathbf{box}\ \|\underline{e}\|
\end{aligned}
$$

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 1

1. *If* $\|e\| = M$ *then:*
   (a) *if* $\mathcal{D}_r :: (\Delta; \Gamma \overset{r}{\vdash} e : \tau)$ *then we have* $\mathcal{D}_i :: (|\Delta|; \|\Gamma\| \overset{i}{\vdash} M : \|\tau\|)$*;*
   (b) *if* $\mathcal{D}_i :: (|\Delta|; \|\Gamma\| \overset{i}{\vdash} M : A)$ *then we have* $\mathcal{D}_r :: (|\Delta|; \|\Gamma\| \overset{r}{\vdash} e : \tau)$ *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 at the 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

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

then applying part *2a* of the induction hypothesis to $\mathcal{D}'_c$ to get $\mathcal{D}'_i$ we get

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

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

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

then applying part *1a* of the induction hypothesis to $\mathcal{D}'_r$ to get $\mathcal{D}'_i$ we get

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

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.

$\Box$

By examining this proof we can verify that the translation of a two-level term can always be type-checked only using the tpi_unbox and tpi_pop rules when tpi_unbox immediately follows tpi_pop. This corresponds to a weaker modal logic, K, in which we drop the assumption in S4 that the accessibility relation is reflexive and transitive [MM94].

In fact, we can define a language Mini-ML$^{\Box}_{K}$ by replacing the **unbox** and **pop** constructors with one equivalent to $\mathbf{unbox}_1$ as in [MM94]. Then, Mini-ML$^{\Box}_{K}$ closely models Mini-ML$_2$, but permits an arbitrary number of phases, each of which can only execute the code generated by the immediately preceding one. This is similar to the idea of B-level languages in [NN92] (with B linearly ordered), and in fact a B-level version of Mini-ML would be exactly equivalent to Mini-ML$^{\Box}_{K}$, by a natural extension of the two-level translation. It is also similar to the Multi-level Generating Extensions of [GJ95].

It is interesting then to consider what the reflexitivity and transitivity assumptions model in the context of staged computation. Essentially they allow us to execute generated code at any future time, or immediately. It would be difficult to achieve the same in an extension of a two-level language, since the separation between the levels is achieved by duplicating the term and type constructors. Hence we consider Mini-ML$^{\Box}$ to be an appropriate language in which to study more general forms of staged computation, including run-time code generation.

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:

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

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

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

Unfortunately, when applied to the first argument, this function generally won't terminate. This is a common problem in partial evaluation, and the usual solution is 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.

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

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:

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

Notice that `acker m'` is unnecessarily calculated twice. This could be avoided if memoization was employed during the compilation.

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$^{\Box}$. Note that we have assumed a data type vector in the example, along with a function $\mathit{sub}{:}\mathsf{nat} \to \mathsf{vector} \to \mathsf{nat}$ to access the elements of a vector.

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

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

We add in $\Box$, **box** and $\mathbf{unbox}_i$ to get a function with three computation stages. 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.

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

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 following non-specializing one, which makes use of a continuation function that is called with the remaining input if the current matching succeeds.

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

(*  val accept' : regexp -> (string list -> bool) -> (string list -> bool)  *)
```

```
fun accept' (Empty) k s = k s
  | accept' (Plus(r1,r2)) k s = accept' r1 k s orelse accept' r2 k s
  | accept' (Times(r1,r2)) k s = accept' r1 (fn s' => accept' r2 k s') s
  | accept' (Star(r)) k s =
      k s orelse
      accept' r (fn s' => if s = s' then false else accept' (Star(r)) k s') s
  | accept' (Const(str)) k (x::s) = (x = str) andalso k s
  | accept' (Const(str)) k (nil) = false

(*  val accept : regexp -> (string list -> bool)  *)

fun accept r s = accept' r (fn nil => true | (x::l) => false) s
```

Note that there is a recursive call to `accept' Star(r)` in the case for `accept' 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:

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

Then, we can add in modal constructors to get our staged program with the following types (using `$` here to represent $\Box$)

```
val accept2' : 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 `unbox_1` for $\mathbf{unbox}_1 \equiv \mathbf{unbox}\ (\mathbf{pop}\ \cdot)$.

```
fun accept2' (Empty) = box (fn k => fn s => k s)
  | accept2' (Plus(r1,r2)) =
        box (fn k => fn s =>
              (unbox_1 (accept2' r1)) k s
              orelse (unbox_1 (accept2' r2)) k s)
  | accept2' (Times(r1,r2)) =
        box (fn k => fn s =>
              (unbox_1 (accept2' r1)) (fn s' => (unbox_1 (accept2' r2)) k s') s)
  | accept2' (Star(r)) =
        box (fn k => fn s =>
              let fun acc2' k s =
```

```
                       k s orelse
                       ((unbox_1 (accept2' r))
                           (fn s' => if s = s' then false else acc2' k s') s)
               in
                 acc' k s
               end)
  | accept2' (Const(str)) =
         box (fn k => (fn s =>
               case s of (x::s') => (x = lift_string str) andalso k s'
                       | nil => false))

fun accept2 r = box (fn s => (unbox_1 (accept2' r))
                              (fn s' => case s' of nil => true | (x::l) => false)
                              s)
```

We can now use our compilation to the explicit language Mini-ML$_e^\Box$ to get an equivalently staged program without the modal operators. We can then further translate to a program in pure Standard ML, which is staged in the same way, but without the modal annotations. It is uneccessary to replace `$A` by `unit -> A` in this case, since `box` is only applied to values. We show this program only to demonstrate the staging described by the the modal annotated program. The program in Mini-ML$_e^\Box$ has the potential to be more efficient, since optimized code can be generated by a sophisticated implementation.

```
(*  val accept3' : regexp -> (string -> bool) -> (string -> bool)  *)

fun accept3' (Empty) = (fn k => fn s => k s)
  | accept3' (Plus(r1,r2)) =
    let val a1 = accept3' r1
        val a2 = accept3' r2
    in
        (fn k => fn s => a1 k s orelse a2 k s)
    end
  | accept3' (Times(r1,r2)) =
    let val a1 = accept3' r1
        val a2 = accept3' r2
    in
        (fn k => fn s => a1 (fn s' => a2 k s') s)
    end
  | accept3' (Star(r1)) =
    let val a1 = accept3' r1
        fun acc2 k s = k s orelse
                       a1 (fn s' => if s = s' then false else acc2 k s') s
    in
        (fn k => fn s => acc2 k s)
    end
  | accept3' (Const(str)) =
```

```
    (fn k => (fn (x::s) => (x = str) andalso k s
               | nil => false))

(*  val accept3 : regexp -> (string -> bool)  *)

fun accept3 r = accept3' r (fn nil => true | (x::l) => false)
```

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. This language is too verbose to be practical, so we continued by defining an implicit language Mini-ML$^\Box$ which, with some syntactic sugar, might serve as the core for an extension of a language with the complexity of Standard ML. The operational semantics of Mini-ML$^\Box$ is given by a compilation to Mini-ML$_e^\Box$. It generalizes Nielson & Nielson's two-level functional language [NN92] which is demonstrated by a conservative embedding theorem, the main technical result of this paper.

The two-level language we consider, Mini-ML$_2$, is directly based on the one in [NN92], but has a stricter binding-time correctness criterion than used, for example, in [GJ91]. Essentially, this restriction may be traced to the fact that our underlying evaluation model applies only to closed terms, while [GJ91] seems to require evaluation of terms with free variables. Glück and Jørgensen [GJ95] present a multi-level binding-time analysis with the less strict binding-time correctness criterion, along with practical motivations for multi-level partial evaluation, though they do not treat higher order functions. A modal operator similar to the "next" operator from temporal logic looks promising as a candidate to model this looser correctness criterion, but we have yet to develop this line of research.

Our language Mini-ML$^\Box$ requires the insertion of the **box**, **unbox** and **pop** coercions into a functional program. It may be preferable for these coercions to remain implicit, though in such a language valid expressions no longer have unique or even principal types, thus raising coherence problems. We intend to study a language in which the modal types are considered refinements of the usual Mini-ML types, using intersections to express principal types (see [FP91] for analogous non-modal refinement types). Refinement type inference for this language would be a form of generalized, polyvariant binding-time analysis. Compilation would be type-directed, generating different versions of functions appropriate for different stagings of computation. The programmer would control this process through refinement type constraints imposed upon functions by type annotations. Type inference in such a language would need to depend strongly on subtyping via implicit coercions between refinement types.

Our operational semantics is also rather naive from a partial evaluation point of view. In particular, we do not memoize during specialization. A memoizing semantics would be desirable for a serious implementation, and would require some restrictions on side-effects. See [BW93] for a description of a serious partial evaluator for Standard ML, which in part inspired this work.

This paper does not treat polymorphism, though it seems that it should not cause any problems. We expect our type system to interact very well with ML's module system. In fact, part of our motivation was to provide the programmer with means to specify staging (= 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 should allow efficient code to be generated within a more 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 plan to carry out more realistic experiments in the near future.

7 Acknowledgements

We gratefully acknowledge discussions with Lars Birkedal, Olivier Danvy, Joëlle Despeyroux, Andrzej Filinski, Karoline Malmkjær, Greg Morrisett, Morten Welinder, and Hao-Chi Wong regarding the subject of this paper.

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. Technical Report DIKU-report 93/22, DIKU, Department of Computer Science, University of Copenhagen, October 1993.

[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.

[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.

[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. Unpublished Manuscript, 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*, volume 523 of *Lecture Notes in Computer Science*, pages 448–472. Springer, Berlin, Heidelberg, New York, 1991.

[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.

[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, To appear.

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

[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. To appear in *Electronic Notes in Theoretical Computer Science*, Volume 1, Elsevier.