A Temporal-Logic Approach to Binding-Time Analysis

Rowan Davies*
Carnegie Mellon University
Computer Science Department
Pittsburgh PA 15213, USA
rowan@cs.cmu.edu

Abstract

The Curry-Howard isomorphism identifies proofs with typed λ-calculus terms, and correspondingly identifies propositions with types. We show how this isomorphism can be extended to relate constructive temporal logic with binding-time analysis. In particular, we show how to extend the Curry-Howard isomorphism to include the $\bigcirc$ ("next") operator from linear-time temporal logic. This yields the simply typed $\lambda^{\bigcirc}$-calculus which we prove to be equivalent to a multi-level binding-time analysis like those used in partial evaluation for functional programming languages. Further, we prove that normalization in $\lambda^{\bigcirc}$ can be done in an order corresponding to the times in the logic, which explains why $\lambda^{\bigcirc}$ is relevant to partial evaluation. We then extend $\lambda^{\bigcirc}$ to a small functional language, Mini-ML$^{\bigcirc}$, and give an operational semantics for it. Finally, we prove that this operational semantics correctly reflects the binding-times in the language, a theorem which is the functional programming analog of time-ordered normalization.

1. Introduction

Partial evaluation [12] is a method for specializing a program given part of the program's input. The basic technique is to execute those parts of the program that do not depend on the unknown data, while constructing a residual program from those parts that do. Offline partial evaluation uses a binding-time analysis to determine those parts of the program that will not depend on the unknown (dynamic) data, regardless of the actual value of the known (static) data.

Binding-time analyses are usually described via typed languages that include binding-time annotations, as for example by Nielson and Nielson [14] and Gomard and Jones [9]. However, the motivation for the particular typing rules that are chosen is often not clear. There has been some work, for example by Palsberg [15], on modular proofs that binding-time analyses generate annotations that allow large classes of partial evaluators to specialize correctly. However this still does not provide a direct motivation for the particular rules used in binding-time type systems.

In this paper we give a logical construction of a binding-time type system based on temporal logic. Temporal logic is an extension of logic to include proofs that formulas are valid at particular times. The Curry-Howard [11] isomorphism relates constructive proofs to typed λ-terms and formulas to types. Thus, we expect that extending the Curry-Howard isomorphism to constructive temporal logic should yield a typed λ-calculus that expresses that a result of a particular type will be available at a particular time. This is exactly what a binding-time type system should capture.

Many different temporal logics and many different temporal operators have been studied, so we need to determine exactly which are relevant to binding-time analysis. In a binding-time separated program, one stage in the program can manipulate as data the code of the following stage. At the level of types this suggests that at each stage we should have a type for code of the next stage. Thus, via the Curry-Howard isomorphism we are led to consider the temporal logic $\bigcirc$ operator, which denotes truth at the next stage, i.e. $\bigcirc A$ is valid if A is valid at the next time. Further, since temporal logics generally allow an unbounded number of "times", they should naturally correspond to a binding-time analysis with many levels, such as that studied by Glück and Jørgensen [8]. The more traditional two-level binding-time analyses can then be trivially obtained by restriction. Also, in binding-time analysis we have a simple linear ordering of the binding times, so we consider linear-time temporal logic, in which each time has a unique time immediately

*This work was mostly completed during a visit by the author to BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation) in the summer of 1995. It was also partly supported by a Hackett Studentship from the University of Western Australia

following it. Putting this all together naturally suggests that constructive linear-time temporal logic with $\bigcirc$ and a type system for multi-level binding-time analysis should be images of each other under the Curry-Howard isomorphism. This does not seem to have been observed previously, and in this paper we show formally that this is indeed the case. The development is relatively straightforward and our main interest is in demonstrating the direct logical relationship between binding-time analysis and temporal logic.

The structure of this paper is then as follows. In the following section, we start with a natural deduction formulation of intuitionistic linear-time temporal logic with $\rightarrow$ and $\bigcirc$. Our system has a similar style to the modal systems of Martini and Masini [13]. We then verify that our $\bigcirc$ operator is the same as that considered elsewhere by showing that adding negation and classical reasoning leads to a system equivalent to $L^{\bigcirc}$, an axiomatic formulation due to Stirling [17] for a small classical linear-time temporal logic including $\bigcirc$.

We then apply the Curry-Howard isomorphism to the natural-deduction system, yielding the typed $\lambda^{\bigcirc}$-calculus with the $\bigcirc$ operator in the types. We give reduction rules for this calculus, and prove that normalization can be performed in an order corresponding to the times in the logic. This theorem gives an abstract explanation for why $\lambda^{\bigcirc}$ is relevant to partial evaluation.

In the second part of the paper we consider $\lambda^{\mathbf{m}}$, which is essentially the λ-calculus fragment of the language used in the multi-level binding-time analysis of Glück and Jørgensen [8]. We then construct a simple isomorphism between $\lambda^{\mathbf{m}}$ and $\lambda^{\bigcirc}$ that preserves typing, thus showing that these languages are equivalent as type systems. We also give a correspondence between β-reductions in the two languages. This gives some explanation for why languages like $\lambda^{\mathbf{m}}$ are relevant to partial evaluation.

Finally, we expand $\lambda^{\bigcirc}$ to a small temporal functional language Mini-ML$^{\bigcirc}$, which is in many ways similar to a realistic binding-time type system. We give an operational semantics for this language, and then give a proof that this semantics correctly reflects the binding times in the language. This theorem is the functional programming analog of the time-ordered normalization theorem for $\lambda^{\bigcirc}$.

We conclude by giving example programs in Mini-ML$^{\bigcirc}$ and discussing some practical concepts from binding-time analysis.

This work is similar to work by Davies and Pfenning [5] which shows that a typed language Mini-ML$^{\Box}$ based on modal logic captures a powerful form of staged computation, including run-time code generation. They also show that Mini-ML$^{\Box}$ is a conservative extension of the two-level and (linearly ordered) B-level languages studied by Nielson and Nielson [14]. However, they note that this system only allows programs that manipulate closed code, while the binding-time type systems used in partial evaluation, such as that of Gomard and Jones [9], allow manipulation of code with free variables. Thus, the original motivation for the present work was to consider how to extend Mini-ML$^{\Box}$ to a system that is a conservative extension of the binding-time type systems used in partial evaluation. In this paper we achieve that goal, though find that we also lose the features of Mini-ML$^{\Box}$ beyond ordinary binding-time analysis. Our conclusion is that the $\Box$ operator in Mini-ML$^{\Box}$ corresponds to a type for *closed code*, while the $\bigcirc$ operator in Mini-ML$^{\bigcirc}$ corresponds to a type for *code with free variables*. Thus the two operators are suitable for different purposes, and which one is preferred depends on the context. This suggests that a desirable direction for future work would be the design of a type system correctly capturing both closed code and code with free variables within a single framework.

2. A temporal λ-calculus

In this section we will show how to extend the Curry-Howard isomorphism to include the $\bigcirc$ ("next") operator from temporal logic. In order to do this, we give a natural-deduction system for an intuitionistic linear-time temporal logic containing only $\bigcirc$ and $\rightarrow$. We then show that our $\bigcirc$ operator is the same as the one usually considered in linear-time temporal logics by adding inference rules for negation and classical reasoning, and then proving equivalence to a sound and complete axiomatic system given by Stirling [17] for the fragment of classical linear-time temporal logic containing only $\bigcirc$, $\rightarrow$ and $\neg$. We then add proof terms to our original natural deduction system to yield a simply typed λ-calculus with the $\bigcirc$ operator in the types.

2.1. Intuitionistic Temporal Logic

This subsection presents a simple natural-deduction formulation of an intuitionistic linear-time temporal logic containing only $\bigcirc$ and $\rightarrow$. Our formulation uses a judgement annotated with a natural number n, representing the "time" of the conclusion, and with each assumption A in Γ also annotated by a time n. These are just like the "levels" in the modal natural-deduction systems of Martini and Masini [13], and in fact our system is exactly the same as their rules for modal K, except that because of linearity we do not need any restriction on the assumptions used in the introduction rule for $\bigcirc$. Our rules for the non-temporal fragment are completely standard.

$$\frac{A^n \text{ in } \Gamma}{\Gamma \vdash^n A} V$$

$$\frac{\Gamma, A_1^n \vdash^n A_2}{\Gamma \vdash^n A_1 \to A_2} \to I$$

$$\frac{\Gamma \vdash^n A_1 \to A_2 \qquad \Gamma \vdash^n A_1}{\Gamma \vdash^n A_2} \to E$$

$$\frac{\Gamma \vdash^{n+1} A}{\Gamma \vdash^n \bigcirc A} \bigcirc I \qquad \frac{\Gamma \vdash^n \bigcirc A}{\Gamma \vdash^{n+1} A} \bigcirc E$$

In order to give a proof-theoretic semantics to the $\bigcirc$ operator we also need to consider a proof reduction rule for $\bigcirc I$ immediately followed by $\bigcirc E$, which reduces trivially to the derivation with both inference steps removed.

2.2. Comparison with a previous temporal logic

We now need to verify that the natural-deduction system presented the previous section really does present the same $\bigcirc$ operator as other linear-time temporal logics. Informally, it is not difficult to see how these rules correspond to a Kripke semantics with a linear reachability relation. However, rather than give a formal proof of soundness and completeness with respect to such a semantics, we instead show equivalence to a previously presented temporal logic.

Unfortunately, intuitionistic temporal logics do no seem to have been considered previously, nor temporal logics without negation. Thus we add classical reasoning and negation to our natural deduction system to make this comparison, even though they are not directly relevant to binding-time analysis. We thus add the following inference rules to those above:

$$\frac{\Gamma, A^n \vdash^n p}{\Gamma \vdash^n \neg A} \neg I^p \qquad (p \text{ a variable not occuring in } \Gamma \text{ or } A)$$

$$\frac{\Gamma \vdash^n A \qquad \Gamma \vdash^n \neg A}{\Gamma \vdash^{n'} B} \neg E$$

$$\frac{\Gamma, \neg A^n \vdash^n A}{\Gamma \vdash^n A} \mathrm{C}$$

These rules are relatively standard, except for the addition of the time annotations.

Note that by presenting the $\bigcirc$ operator using only an introduction rule and elimination rule in the original system we have separated it from negation and classical reasoning.

We are now in a position to prove equivalence to a previously presented temporal logic. The following axioms and inference rules, $L^{\bigcirc}$, for the fragment of classical linear-time temporal logic containing only $\bigcirc$, $\to$ and $\neg$ are due to Stirling [17] (page 516), who shows that they are sound and complete for unravelled models of this logic. We choose this system as our starting point because it appears to be the smallest linear temporal logic containing the $\bigcirc$ operator that has been previously considered in the literature.

Axioms:	L1	Any classical tautology instance
	L3	$\bigcirc\neg A \;\leftrightarrow\; \neg\bigcirc A$
	L4	$\bigcirc(A_1 \to A_2) \;\to\; (\bigcirc A_1 \to \bigcirc A_2)$
Inference rules:	MP	if $A_1 \to A_2$ and A_1 then A_2
	RO	if A then $\bigcirc A$

Note that in the inference rules, we require that there be proofs from no assumptions.

The following theorem shows that our $\bigcirc$ operator is the same as the one usually considered in linear-time temporal logic.

Theorem 1 *We can derive $\cdot \vdash^0 A$ in the extended natural-deduction system if and only if there is a proof of A in $L^{\bigcirc}$.*

Proof: (sketch) In one direction, we construct a derivation for each axiom and proceed by induction over the inference rules in $L^{\bigcirc}$. For L1 we actually simply notice that removing the rules for $\bigcirc$ yields a standard natural-deduction system for pure classical logic. The other axioms are straightforward, and the case for the inference rule RO only requires showing that incrementing every time annotation in a derivation yields a derivation.

We prove the other direction by induction over the structure of derivations, strengthening the induction hypothesis to:

if $A_1^{n_1}, \ldots, A_k^{n_k} \vdash^n A$
then $\bigcirc^{n_1} A_1 \to \ldots \to \bigcirc^{n_k} A_k \to \bigcirc^n A$
is provable in $L^{\bigcirc}$

Here $\bigcirc^n$ means n occurrences of $\bigcirc$. Only the cases for the $\to$ and $\neg$ rules are non-trivial. They are solved by repeated application of L3, L4 and the converse of L4 (which is derivable using L3, L4 and a classical tautology) along with a sequence of cuts (which are classical tautologies).

For example, in the case for $\to I$, we have by the induction hypothesis that

$$\bigcirc^n A'_1 \to \ldots \to \bigcirc^n A'_m \to \bigcirc^n A_1 \to \bigcirc^n A_2$$

We can also deduce that

$$(\bigcirc^n A_1 \to \bigcirc^n A_2) \to \bigcirc^n (A_1 \to A_2)$$

by repeated application of the converse of L4.

Now, by using the cut-formula (which is a classical tautology)

$$\begin{array}{c}(\bigcirc^n A'_1 \to \ldots \to \bigcirc^n A'_m \to \bigcirc^n A_1 \to \bigcirc^n A_2)\\ \to ((\bigcirc^n A_1 \to \bigcirc^n A_2) \to \bigcirc^n (A_1 \to A_2))\\ \to (\bigcirc^n A'_1 \to \ldots \to \bigcirc^n A'_m \to \bigcirc^n (A_1 \to A_2))\end{array}$$

and MP twice, we get the required result.

□

2.3. A temporal λ-calculus

We now add proof terms to the original intuitionistic temporal-logic natural-deduction system. This yields $\lambda^{\bigcirc}$, a simply typed λ-calculus with the $\bigcirc$ operator in the types, by the natural extension of the Curry-Howard isomorphism.

2.3.1 Syntax

$$\begin{array}{llcll}\text{Types} & A & ::= & b & \mid A_1 \to A_2 \mid \bigcirc A\\ \text{Terms} & M & ::= & x & \mid \lambda x{:}A.\ M \mid M_0\, M_1\\ & & & & \mid \textbf{next}\ M \mid \textbf{prev}\ M\\ \text{Contexts} & \Gamma & ::= & \cdot & \mid \Gamma, x{:}A^n\end{array}$$

2.3.2 Typing rules

$\Gamma \vdash^n M : A$ — Expression M has type A at time n in context Γ.

$$\frac{x{:}A^n \text{ in } \Gamma}{\Gamma \vdash^n x : A}\, V$$

$$\frac{\Gamma, x{:}A_1^n \vdash^n M : A_2}{\Gamma \vdash^n \lambda x{:}A_1.\ M : A_1 \to A_2} \to I$$

$$\frac{\Gamma \vdash^n M_0 : A_1 \to A_2 \qquad \Gamma \vdash^n M_1 : A_1}{\Gamma \vdash^n M_0\, M_1 : A_2} \to E$$

$$\frac{\Gamma \vdash^{n+1} M : A}{\Gamma \vdash^n \textbf{next}\ M : \bigcirc A} \bigcirc I \qquad \frac{\Gamma \vdash^n M : \bigcirc A}{\Gamma \vdash^{n+1} \textbf{prev}\ M : A} \bigcirc E$$

2.3.3 Reduction rules

We have the standard β-reduction rule as well as the following temporal reduction rules. The first comes from the natural-deduction proof-reduction rule, analogously to the correspondence between β-reduction and the proof reduction rule for $\to$:

$$\textbf{prev}(\textbf{next}\ M) \xrightarrow{\bigcirc} M$$

We also have the following for elimination followed by introduction, analogous to η-reduction:

$$\textbf{next}(\textbf{prev}\ M) \xrightarrow{\bigcirc} M$$

2.3.4 Time-ordered normalization

Suppose we have $\Gamma \vdash^n M : A$. Then we say that the typing derivation associates a time n' with the subterm M' of M if the corresponding sub-derivation of the typing derivation has the form $\Gamma' \vdash^{n'} M' : A'$. Note that reduction preserves the time of subterms. Also, since β-redices are sub-terms, this definition associates a time with each one, and this time will be the same as that associated with both the body of the λ and the argument.

Now, a fundamental property of $\lambda^{\bigcirc}$ is that we can reduce β-redices in the order of their associated time, provided we reduce all temporal redices between β-reductions. In doing so, each reduction cannot lead to β-redices with an earlier time. We refer to this property as "time-ordered normalization", and prove it below.

This property explains why $\lambda^{\bigcirc}$ is relevant to partial evaluation. If we have a term in $\lambda^{\bigcirc}$, then we know that β-reduction can be done in stages corresponding to the times in the temporal logic. This is very closely related to binding-time correctness in a functional language, which we will discuss in the next section.

Theorem 2 (Time-ordered normalization) *Suppose we have $\Gamma \vdash^n M : A$, where M has no temporal redices and no β-redices with time less than n'. Then reducing a β-redex with time n' and then reducing all temporal redices leads a term which also has no β-redices with time less than n'.*

In the proof we make use of the standard notation $C[M']$ to indicate the term where M' appears in the context C.

Proof: We have $M \equiv C[(\lambda x{:}A.\ M_0)M_1]$ with $\Gamma', x{:}A^{n'} \vdash^{n'} M_0 : A_0$ and $\Gamma' \vdash^{n'} M_1 : A_1$. We reduce to $M' \equiv C[\{M_1/x\}M_0]$ and then reduce all temporal redices.

Suppose there is a β-redex with time less than n' in the result. Then we have one of two cases:

1. $M' \equiv C'[C''[\{M_1/x\}M_0]M_2]$ with $C''[\{M_1/x\}M_0]$ reducing to $\lambda y{:}A.\ M_0'$ by temporal reductions. But then C'' cannot be the identity since then the resulting β-redex has time n'. Nor can it begin with a λ, since then there is β-redex with time less than n' in M. Also, by the typing, C'' cannot begin with **next**. Hence, it must begin with **prev**. Continuing this argument, we see that $C'' \equiv \textbf{prev}^i$. But, then $[M_1/x]M_0 \equiv \textbf{next}^i(\lambda t{:}A.\ M_0')$, and the resulting redex has level $n' + i$, which is not less than n'.

2. $[M_1/x]M_0 \equiv C'[C''[M_1]M_2]$ with $C''[M_1]$ reducing to $\lambda y{:}A.\ M_1'$ by temporal reductions. By an almost identical argument to the above, this is impossible. □

3. Equivalence to a binding-time type system

We now demonstrate the relationship between $\lambda^{\bigcirc}$ and binding-time type systems considered by other authors. To do this we consider $\lambda^{\mathbf{m}}$, a simply typed λ-calculus which is essentially the core of standard binding-time analyzes used in offline partial evaluation (see e.g. Gomard and Jones [9]). $\lambda^{\mathbf{m}}$ additionally allows more than two binding times in the same way as the multi-level binding-time analysis of Glück and Jørgensen [8]. Our formulation of $\lambda^{\mathbf{m}}$ is basically the λ-calculus fragment of Glück and Jørgensen's system, though it has some important differences. We use separate syntactic categories for the types of each level, thus avoiding side conditions regarding well-formedness of types. Further, we do not treat the final level as dynamically typed, but consider the whole program to be statically typed. Finally, we do not include "lifting" from one binding time to a later one, but instead demonstrate later how this can be easily added.

We then give a simple translation between $\lambda^{\mathbf{m}}$ and $\lambda^{\bigcirc}$. This translation is a bijection on terms and types that preserves typing, modulo reduction of temporal redices. Thus $\lambda^{\mathbf{m}}$ and $\lambda^{\bigcirc}$ are equivalent as type systems, modulo these trivial reductions.

3.1. Syntax

We use the separate syntactic categories τ^n to indicate the type of results which will be available at time n or later. The time annotations on base types b^n and function types $\tau_1^n \to^n \tau_2^n$ indicate the time at which the corresponding values are available. The time annotations on terms indicate the time at which the λ or @ (application) are reduced, and the corresponding variable substituted for. See Glück and Jørgensen [8] for a semantics of evaluation of multi-level terms in multiple stages.

$$
\begin{array}{llll}
\text{Types} & \tau^n & ::= & b^n \mid \tau_1^n \xrightarrow{n} \tau_2^n \mid \tau^{n+1} \\
\text{Terms} & E & ::= & x^n \mid \lambda^n x^n{:}\tau^n.\ E \mid E_0 \ @^n\ E_1 \\
\text{Contexts} & \Psi & ::= & \cdot \mid \Psi, x^n{:}\tau^n
\end{array}
$$

3.2. Typing rules

$$\frac{x^n{:}\tau^n \text{ in } \Psi}{\Psi \vdash^{\mathbf{m}} x^n : \tau^n}\ \mathsf{tpm\_var}$$

$$\frac{\Psi, x^n{:}\tau_1^n \vdash^{\mathbf{m}} E : \tau_2^n}{\Psi \vdash^{\mathbf{m}} \lambda^n x^n{:}\tau^n.\ E : \tau_1^n \xrightarrow{n} \tau_2^n}\ \mathsf{tpm\_lam}$$

$$\frac{\Psi \vdash^{\mathbf{m}} E_0 : \tau_1^n \xrightarrow{n} \tau_2^n \qquad \Psi \vdash^{\mathbf{m}} E_1 : \tau_1^n}{\Psi \vdash^{\mathbf{m}} E_0 \ @^n\ E_1 : \tau_2^n}\ \mathsf{tpm\_app}$$

3.3. Equivalence translation

We now give simple translations between well typed terms in $\lambda^{\mathbf{m}}$ and $\lambda^{\bigcirc}$. The translation from $\lambda^{\bigcirc}$ maps terms which are in the same equivalence class with respect to temporal reductions to the same $\lambda^{\mathbf{m}}$ term. In the other direction, we always translate $\lambda^{\mathbf{m}}$ terms to a $\lambda^{\bigcirc}$ terms with no temporal redices, these being unique representatives of the equivalence classes. Note that we can always reduce all temporal redices, since the number of reductions is bounded by the number of **next** and **prev** constructors. We then show that the two translations preserve typing and are inverses of each other when restricted to $\lambda^{\bigcirc}$ terms with no temporal redices. This shows that $\lambda^{\mathbf{m}}$ is isomorphic to $\lambda^{\bigcirc}$ modulo the temporal reductions.

The translations are given by the functions $|\cdot|^n$ and $||\cdot||^n$ defined as follows:

Type Translations

$$
\begin{array}{rcl}
|b^n|^n & = & b \\
|\tau_1^n \to^n \tau_2^n|^n & = & |\tau_1^n|^n \to |\tau_2^n|^n \\
|\tau^{n+1}|^n & = & \bigcirc|\tau^{n+1}|^{n+1}
\end{array}
$$

$$
\begin{array}{rcl}
||b||^n & = & b^n \\
||A_1 \to A_2||^n & = & ||A_1||^n \to^n ||A_2||^n \\
||\bigcirc A||^n & = & ||A||^{n+1}
\end{array}
$$

Term Translations

$$
\begin{array}{rl}
|x^n|^n & = x \\
|\lambda^n x^n{:}\tau^n.\ E|^n & = \lambda x{:}|\tau^n|^n.\ |E|^n \\
|E_0 @^n E_1|^n & = |E_0|^n\ |E_1|^n \\
(m > n) \quad |E^m|^n & = \mathbf{next}|E^m|^{n+1} \\
(m < n+1)\ |E^m|^{n+1} & = \mathbf{prev}|E^m|^n
\end{array}
$$

$$
\begin{array}{rl}
||x||^n & = x^n \\
||\lambda x{:}A.\ M||^n & = \lambda x^n{:}||A^n||^n.\ ||M||^n \\
||M_0\ M_1||^n & = ||M_0||^n @^n ||M_1||^n \\
||\mathbf{next}\ M||^n & = ||M||^{n+1} \\
||\mathbf{prev}\ M||^{n+1} & = ||M||^n
\end{array}
$$

Here we use E^m as convenient syntax matching all expressions which have the top constructor annotated with m.

Lemma 3 *$|\cdot|^n$ and $||\cdot||^n$ are inverses on the fragment of $\lambda^{\bigcirc}$ with no* **next**-**prev** *redices.*

Proof: By a straightforward induction. □

Theorem 4 *The two translations preserve typing, namely:*

- *if* $\cdot \vdash^{\mathbf{m}} E : \tau^0$ *then* $\cdot \vdash^0 |M|^0 : |\tau^0|^0$
- *if* $\cdot \vdash^0 M : A$ *then* $\cdot \vdash^{\mathbf{m}} ||M||^0 : ||A||^0$

Proof: By a straightforward structural induction, strengthening appropriately to:

- if $\Psi \vdash^{\mathbf{m}} E : \tau^n$ then $|\Psi| \vdash^n |M|^n : |\tau|^n$
- if $\Gamma \vdash^n M : A$ then $||\Gamma|| \vdash^{\mathbf{m}} ||M||^n : ||A||^n$

$\square$

Lemma 5 *If* $||M||^n = E$ *then* $M \xrightarrow{\bigcirc} |E|^n$ *using only temporal reductions.*

Proof: By induction on M. $\square$

We can now define β-reduction in $\lambda^{\mathbf{m}}$ as the image of β-reduction in $\lambda^{\bigcirc}$ under the translation, namely:

$$(\lambda^n x^n{:}\tau^n.\, E_0) @^n E_1 \xrightarrow{\beta} [E_1/x^n]E_0$$

Then, using the above lemma, if $||M||^n = E$ and $E \xrightarrow{\beta} E'$ then $M \xrightarrow{\bigcirc} M_1$ using only temporal reductions, and $M_1 \xrightarrow{\beta} M'$ with $||M'||^n = E'$. Informally, this means that β reductions correspond exactly between $\lambda^{\bigcirc}$ and $\lambda^{\mathbf{m}}$.

At this stage, we can consider $\lambda^{\bigcirc}$ and $\lambda^{\mathbf{m}}$ to be equivalent except for the temporal reductions in $\lambda^{\bigcirc}$ which are hidden by the syntax in $\lambda^{\mathbf{m}}$. Thus $\lambda^{\bigcirc}$ expresses binding-times in the same way as $\lambda^{\mathbf{m}}$. Conversely, $\lambda^{\mathbf{m}}$ can be considered as proof terms for our temporal logic, with an alternative, equivalent syntax for formulas. This justifies our claim that the core of binding-time type systems are the image of a temporal logic under the the Curry-Howard isomorphism.

4. A temporal functional language

We now show how to extend $\lambda^{\bigcirc}$ to obtain Mini-ML$^{\bigcirc}$, a small functional language in the style of [3], which includes **next**, **prev** and $\bigcirc$ to allow expression of binding-times. We give an operational semantics for this language, and then prove type preservation and value soundness. We also demonstrate that this operational semantics correctly reflects the binding-times.

Often in partial evaluation the binding-time type system is only used to generate binding-time information which guides a specializer, and no operational semantics is given directly to the binding-time language. Here we consider that the operational semantics is fundamental in that it gives meaning to the binding-times. We claim that other methods for interpreting the binding-times can generally be formulated as binding-time preserving transformations in a language like Mini-ML$^{\bigcirc}$. For example, the generation of specialization points in a partial evaluator like Similix (see [1] or [12]) can be expressed as adding memoizing functions to the binding-time separated program. The semantics of the binding-time analyzed program is then the composition of these binding-time preserving transformations with a semantics like the one we give.

We could similarly extend $\lambda^{\mathbf{m}}$ instead of $\lambda^{\bigcirc}$ to get a functional language syntactically very similar to other binding-time type systems. We choose instead to start with $\lambda^{\bigcirc}$ because we would like to have a language which is syntactically suitable for manually programming with binding-times. A language based on $\lambda^{\mathbf{m}}$ would require the programmer to annotate every construct in a program with a time, which would be very tedious and make programs hard to read. Normally binding-time type systems are only used an output language for a binding-time analysis, so ease of programming isn't an issue. Mini-ML$^{\bigcirc}$ is suitable for manually programming with binding-times, as the examples at the end of this section demonstrate. It is interesting to note that **next** and **prev** are very similar to the quasi-quoting mechanism in the popular programming language Lisp, thus further supporting the claim that they are natural way for the programmer to provide binding-time information.

The motivation for supporting manual programming is to avoid an almost universal problem in automatic partial evaluation, namely that generation of code is not guaranteed to terminate. By allowing the programmer to have direct control over binding-times in a language with a well defined semantics, we can place the responsibility of ensuring termination on the programmer. Welinder [18] has demonstrated that this style of programming, called hand-writing generating extensions, can be very fruitful. Some of the benefits of automatic binding-time analysis might then be regained by allowing the temporal term constructors to be implicit coercions in a system of sub-typing, in the style of Breazu-Tannen *et.al.* [2], leading to a refinement type system [6] where each refinement of a type is obtained by some insertions of $\bigcirc$. The practicality of this idea has yet to be properly investigated.

Our language Mini-ML$^{\bigcirc}$ includes pairs, natural numbers, and fixed-points. We omit polymorphism, but claim that it can be added with some minor complications to the proofs that follow. Mini-ML$^{\bigcirc}$ is explicitly typed, since we do not treat type inference here, although note that type inference for **next** and **prev** can be handled easily using unification.

4.1. Syntax

$$
\begin{array}{llll}
\text{Types} & A & ::= & \mathsf{nat} \mid A_1 \to A_2 \mid A_1 \times A_2 \mid \bigcirc A \\
\text{Terms} & e & ::= & x \mid \lambda x{:}A.\ e \mid e_1\, e_2 \\
& & & \mid \textbf{fix}\ x{:}A.\ e \\
& & & \mid \langle e_1, e_2\rangle \mid \textbf{fst}\ e \mid \textbf{snd}\ e \\
& & & \mid \textbf{z} \mid \textbf{s}\ e \\
& & & \mid (\textbf{case}\ e_1\ \textbf{of}\ \textbf{z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3) \\
& & & \mid \textbf{next}\ e \mid \textbf{prev}\ e \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A^n
\end{array}
$$

4.2. Typing Rules

In this section we present typing rules for Mini-ML$^{\bigcirc}$. The typing judgement has the form:

$\Gamma \vdash^n e : A$ Term e has type A at time n in context Γ.

λ-calculus Fragment

$$\frac{x{:}A^n \text{ in } \Gamma}{\Gamma \vdash^n x : A}\,\mathsf{tpt\_var} \qquad \frac{\Gamma, x{:}A^n \vdash^n e : B}{\Gamma \vdash^n \lambda x{:}A.\ e : A \to B}\,\mathsf{tpt\_lam}$$

$$\frac{\Gamma \vdash^n e_0 : A \to B \qquad \Gamma \vdash^n e_1 : A}{\Gamma \vdash^n e_0\, e_1 : B}\,\mathsf{tpt\_app}$$

Mini-ML Fragment

$$\frac{\Gamma, x{:}A^n \vdash^n e : A}{\Gamma \vdash^n \textbf{fix}\ x{:}A.\ e : A}\,\mathsf{tpt\_fix}$$

$$\frac{\Gamma \vdash^n e_1 : A_1 \qquad \Gamma \vdash^n e_2 : A_2}{\Gamma \vdash^n \langle e_1, e_2\rangle : A_1 \times A_2}\,\mathsf{tpt\_pair}$$

$$\frac{\Gamma \vdash^n e : A_1 \times A_2}{\Gamma \vdash^n \textbf{fst}\ e : A_1}\,\mathsf{tpt\_fst} \qquad \frac{\Gamma \vdash^n e : A_1 \times A_2}{\Gamma \vdash^n \textbf{snd}\ e : A_2}\,\mathsf{tpt\_snd}$$

$$\frac{}{\Gamma \vdash^n \textbf{z} : \mathsf{nat}}\,\mathsf{tpt\_z} \qquad \frac{\Gamma \vdash^n e : \mathsf{nat}}{\Gamma \vdash^n \textbf{s}e : \mathsf{nat}}\,\mathsf{tpt\_s}$$

$$\frac{\Gamma \vdash^n e_1 : \mathsf{nat} \quad \Gamma \vdash^n e_2 : A \quad \Gamma, x{:}\mathsf{nat}^n \vdash^n e_3 : A}{\Gamma \vdash^n (\textbf{case}\ e_1\ \textbf{of}\ \textbf{z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3) : A}\,\mathsf{tpt\_case}$$

Temporal Fragment

$$\frac{\Gamma \vdash^{n+1} e : A}{\Gamma \vdash^n \textbf{next}\, e : \bigcirc A}\,\mathsf{tpt\_next} \qquad \frac{\Gamma \vdash^n e : \bigcirc A}{\Gamma \vdash^{n+1} \textbf{prev}\, e : A}\,\mathsf{tpt\_prev}$$

4.3. Operational Semantics

We now present the operational semantics of Mini-ML$^{\bigcirc}$. The intuition for this semantics is that parts of the term with time 0, the current time, should be evaluated now, while those at later times should be delayed until then. This leads to two rules for each non-temporal construct in the language. The values, v^0, are the standard ones for a Mini-ML like language, with the addition of **next** v^1, where v^1 includes all expressions with no sub-term at time 0. The operational semantics allows for manipulation of expressions containing free variables, provided the variables have time greater than 0, and we thus depend on the fact that substitution avoids variable capture.

The operational semantics for $\lambda^{\bigcirc}$ is similar to the specialization logic presented by Hatcliff [10] for a two-level binding-time language, and our binding-time correctness theorem is also somewhat similar to Hatcliff's.

Values

$$
\begin{array}{lll}
v^0 & ::= & \lambda x{:}A.\ e \mid \langle v_1^0, v_2^0\rangle \mid \textbf{z} \mid \textbf{s}\ v^0 \mid \textbf{next}\ v^1 \\
v^{n+1} & ::= & x \mid \lambda x{:}A.\ v^{n+1} \mid v_1^{n+1}\ v_2^{n+1} \\
& & \mid \textbf{fix}\ x{:}A.\ v^{n+1} \\
& & \mid \langle v_1^{n+1}, v_2^{n+1}\rangle \mid \textbf{fst}\ v^{n+1} \mid \textbf{snd}\ v^{n+1} \\
& & \mid \textbf{z} \mid \textbf{s}\ v^{n+1} \\
& & \mid (\textbf{case}\ v_1^{n+1}\ \textbf{of}\ \textbf{z} \Rightarrow v_2^{n+1} \mid \textbf{s}\ x \Rightarrow v_3^{n+1}) \\
& & \mid \textbf{next}\ v^{n+2} \\
& & \mid \textbf{prev}\ v^n\ \ (n > 0)
\end{array}
$$

$e \overset{n}{\hookrightarrow} v$ Expression e with time n evaluates to value v.

λ-calculus Fragment, time 0

$$\frac{}{\lambda x{:}A.\ e \overset{0}{\hookrightarrow} \lambda x{:}A.\ e}\,\mathsf{ev\_lam}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.\ e_1' \quad e_2 \overset{0}{\hookrightarrow} v_2 \quad [v_2/x]e_1' \overset{0}{\hookrightarrow} v}{e_1\ e_2 \overset{0}{\hookrightarrow} v}\,\mathsf{ev\_app}$$

λ-calculus Fragment, time n+1

$$\frac{}{x \overset{n+1}{\hookrightarrow} x}\,\mathsf{ev\_var'}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\lambda x{:}A.\ e \overset{n+1}{\hookrightarrow} \lambda x{:}A.\ v}\,\mathsf{ev\_lam'}$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \qquad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2}\,\mathsf{ev\_app'}$$

Mini-ML Fragment, time 0

$$\frac{[\mathbf{fix}\ x.\ e/x]e \overset{0}{\hookrightarrow} v}{\mathbf{fix}\ x.\ e \overset{0}{\hookrightarrow} v}\ \textsf{ev\_fix}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad e_2 \overset{0}{\hookrightarrow} v_2}{\langle e_1, e_2\rangle \overset{0}{\hookrightarrow} \langle v_1, v_2\rangle}\ \textsf{ev\_pair}$$

$$\frac{e \overset{0}{\hookrightarrow} \langle v_1, v_2\rangle}{\mathbf{fst}\ e \overset{0}{\hookrightarrow} v_1}\ \textsf{ev\_fst} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v_1, v_2\rangle}{\mathbf{snd}\ e \overset{0}{\hookrightarrow} v_2}\ \textsf{ev\_snd}$$

$$\frac{}{\mathbf{z} \overset{0}{\hookrightarrow} \mathbf{z}}\ \textsf{ev\_z} \qquad \frac{e \overset{0}{\hookrightarrow} v}{\mathbf{s}\ e \overset{0}{\hookrightarrow} \mathbf{s}\ v}\ \textsf{ev\_s}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \mathbf{z} \qquad e_2 \overset{0}{\hookrightarrow} v}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \overset{0}{\hookrightarrow} v}\ \textsf{ev\_case\_z}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \mathbf{s}\ v_1' \qquad [v_1'/x]e_3 \overset{0}{\hookrightarrow} v}{(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \overset{0}{\hookrightarrow} v}\ \textsf{ev\_case\_s}$$

Mini-ML Fragment, time n+1

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{fix}\ x.\ e \overset{n+1}{\hookrightarrow} \mathbf{fix}\ x.\ v}\ \textsf{ev\_fix}'$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \qquad e_2 \overset{n+1}{\hookrightarrow} v_2}{\langle e_1, e_2\rangle \overset{n+1}{\hookrightarrow} \langle v_1, v_2\rangle}\ \textsf{ev\_pair}'$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{fst}\ e \overset{n+1}{\hookrightarrow} \mathbf{fst}\ v}\ \textsf{ev\_fst}' \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{snd}\ e \overset{n+1}{\hookrightarrow} \mathbf{snd}\ v}\ \textsf{ev\_snd}'$$

$$\frac{}{\mathbf{z} \overset{n+1}{\hookrightarrow} \mathbf{z}}\ \textsf{ev\_z}' \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{s}\ e \overset{n+1}{\hookrightarrow} \mathbf{s}\ v}\ \textsf{ev\_s}'$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2 \quad e_3 \overset{n+1}{\hookrightarrow} v_3}{\begin{array}{c}(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3) \overset{n+1}{\hookrightarrow} \\ (\mathbf{case}\ v_1\ \mathbf{of}\ \mathbf{z} \Rightarrow v_2 \mid \mathbf{s}\ x \Rightarrow v_3)\end{array}}\ \textsf{ev\_case}'$$

Temporal Fragment

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{next}\ e \overset{n}{\hookrightarrow} \mathbf{next}\ v}\ \textsf{ev\_next}$$

$$\frac{e \overset{0}{\hookrightarrow} \mathbf{next}\ v}{\mathbf{prev}\ e \overset{1}{\hookrightarrow} v}\ \textsf{ev\_prev}$$

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathbf{prev}\ e \overset{n+2}{\hookrightarrow} \mathbf{prev}\ v}\ \textsf{ev\_prev}'$$

Theorem 6 (Determinacy)

1. *If $e \overset{n}{\hookrightarrow} v$ then v has the form v^n.*

2. *If $e \overset{n}{\hookrightarrow} v$ and $e \overset{n}{\hookrightarrow} v'$ then $v = v'$ (modulo renaming of bound variables).*

Proof: By straightforward inductions over the structure of the derivation of $e \overset{n}{\hookrightarrow} v$. □

Theorem 7 (Type and Time Preservation)

1. *If $e \overset{n}{\hookrightarrow} v$ and $\Gamma \vdash^n e : A$ with Γ not containing any bindings of variables at time 0, then $\Gamma \vdash^n v : A$.*

2. *Further, if a term constructor in e has time n' associated with it by the original typing proof and it gives rise to a term constructor in v (via substitution, the rule* ev_lam*, or a rule for time m+1) then that term constructor has time n' associated with it by the typing proof for v.*

Proof: By a straightforward induction over the structure of the derivation of $e \overset{n}{\hookrightarrow} v$, using an appropriate substitution lemma. □

Lemma 8 *If $\Gamma \vdash^{n+1} v^{n+1} : A$ with Γ not containing any bindings of variables at time 0, then $\Gamma' \vdash^n v^n : A$, where Γ' is obtained from Γ by reducing the time on each variable binding by one.*

Proof: By induction on the structure of the typing derivation. □

Theorem 9 (Binding-time correctness) *If $\cdot \vdash^0 e : \bigcirc A$ and $e \overset{0}{\hookrightarrow} v$ then v has the form* **next** *v' and $\cdot \vdash^0 v' : A$.*

Proof: By type preservation, $\cdot \vdash^0 v : \bigcirc A$. But then **next** v' is the only form that the value v could take in this typing derivation. Then, applying the above lemma gives $\cdot \vdash^0 v' : A$. □

Binding-time correctness in Mini-ML$^{\bigcirc}$ is very closely related to time-ordered normalization in $\lambda^{\bigcirc}$. The binding-time correctness theorem means that we can evaluate a program in Mini-ML$^{\bigcirc}$ with type $\bigcirc^n A$ in $n+1$ stages corresponding to the times in the typing derivation, and if all stages terminate the result will have type A. Each stage except the last involves evaluating the term to a value of the form **next** v^1 and then continuing the next stage with v^1.

Now, by the time preservation property, at stage i the constructors with time 0 (under the associated typing proof) are images of those with time i in the original term, and these are the ones for which real evaluation occurs during the stage. Further, the result of each step except the last has the form **next** v^1, which cannot contain any constructors with time 0 and so all evaluation corresponding to sub-terms with time i has been done. Thus, our operational semantics correctly captures binding-times.

4.4 Examples

We now give some examples of programs in Mini-ML$^{\bigcirc}$. The first is a very common toy example from partial evaluation, namely the power function where the exponent has an earlier binding-time than the index. We give two different versions and compare them. In both we assume a function *times* : nat → nat → nat which multiplies two numbers.

$$
\begin{array}{l}
\mathit{power} \equiv \textbf{fix}\ p{:}\mathsf{nat} \to \bigcirc(\mathsf{nat} \to \mathsf{nat}).\\
\quad \lambda n{:}\mathsf{nat}.\ \textbf{case}\ n\\
\quad\quad \textbf{of z} \Rightarrow \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \textbf{s z})\\
\quad\quad \mid\ \textbf{s}\ m \Rightarrow \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\\
\quad\quad\quad\quad (\textbf{prev}\ \ (p\ m)\ x))
\end{array}
$$

$$
\begin{array}{rl}
\mathit{power}\ \textbf{z} \hookrightarrow & \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \textbf{s z})\\
\mathit{power}\ (\textbf{s z}) \hookrightarrow & \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ ((\lambda x{:}\mathsf{nat}.\ \textbf{s z})x))\\
\mathit{power}\ (\textbf{s}\ (\textbf{s z})) \hookrightarrow & \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\\
& \quad ((\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ ((\lambda x{:}\mathsf{nat}.\ \textbf{s z})x))x))
\end{array}
$$

$$
\begin{array}{l}
\mathit{power}' \equiv \lambda n{:}\mathsf{nat}.\ \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \textbf{prev}\ (\\
\quad (\textbf{fix}\ p{:}\mathsf{nat} \to \bigcirc\mathsf{nat}.\\
\quad\ \lambda n'{:}\mathsf{nat}.\ \textbf{case}\ n'\\
\quad\quad \textbf{of z} \Rightarrow \textbf{next}\ (\textbf{s z})\\
\quad\quad \mid\ \textbf{s}\ m \Rightarrow \textbf{next}\ (\ \mathit{times}\ x\\
\quad\quad\quad\quad (\textbf{prev}\ \ (p\ m)))) \ n))
\end{array}
$$

$$
\begin{array}{rl}
\mathit{power}'\ \textbf{z} \hookrightarrow & \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \textbf{s z})\\
\mathit{power}'\ (\textbf{s z}) \hookrightarrow & \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ (\textbf{s z}))\\
\mathit{power}'\ (\textbf{s}\ (\textbf{s z})) \hookrightarrow & \textbf{next}\ (\lambda x{:}\mathsf{nat}.\ \mathit{times}\ x\ (\mathit{times}\ x\ (\textbf{s z})))
\end{array}
$$

The first version corresponds exactly to the one given by Davies and Pfenning [5] for the functional language Mini-ML$^{\Box}$ based on the modal logic S4. The second avoids the variable for variable redices in the result of applying to the first argument. This program has no counterpart in Mini-ML$^{\Box}$, since it requires evaluation with code containing free variables, which Mini-ML$^{\bigcirc}$ allows but Mini-ML$^{\Box}$ does not. This illustrates the main reason why Mini-ML$^{\bigcirc}$ is interesting compared to Mini-ML$^{\Box}$. However, note that the Mini-ML$^{\Box}$ has other features not supported by Mini-ML$^{\bigcirc}$, namely an operator expressing immediate evaluation of code, and sharing of code between stages. As an example of this, the staged inner product example in [5] has no counterpart in Mini-ML$^{\bigcirc}$.

Abstractly, the reason for these differences is that Mini-ML$^{\Box}$ only allows manipulation of closed code, thus disallowing some programs, while Mini-ML$^{\bigcirc}$ allows manipulation of code with free variables, thus making immediate evaluation of code and sharing of code between stages unsafe. Even more abstractly, the reason for these differences is that the Kripke semantics of the modal logic S4 on which Mini-ML$^{\Box}$ is based is reflexive and transitive, while the Kripke semantics of the temporal logic on which Mini-ML$^{\bigcirc}$ is based is linear.

As another example program in Mini-ML$^{\bigcirc}$, we note that realistic binding-time analyses include the automatic insertion of a "lift" operator at least at base types. The "lift" operator is essentially a coercion from one time to a later one. We now show how to define a function in $\lambda^{\bigcirc}$ with type nat → ◯nat which performs this coercion, exactly following [5]:

$$
\begin{array}{l}
\mathit{lift}_{\mathsf{nat}} \equiv \textbf{fix} f{:}\mathsf{nat} \to \bigcirc\mathsf{nat}.\\
\quad \lambda x{:}\mathsf{nat}.\ \textbf{case}\ x\\
\quad\quad \textbf{of z} \Rightarrow \textbf{next z}\\
\quad\quad \mid\ \textbf{s}\ x' \Rightarrow \textbf{next}\ (\textbf{s}\ (\textbf{prev}\ (f\ x')))
\end{array}
$$

A similar term of type $A \to \bigcirc A$ that returns a **next**'ed copy of its argument will generally exist for each base type, and inductively for pairs. This justifies the inclusion of the *lift* primitive for base types in binding-time type systems such as that of Gomard and Jones [9] and, in a more realistic version of our language, we would also include it as a primitive. Note that we also have the following term with type $(\bigcirc A \to \bigcirc B) \to \bigcirc(A \to B)$ which allows a form of lift on functions:

$$\lambda f{:} \bigcirc A \to \bigcirc B.\ \textbf{next}\ (\lambda x{:}A.\ \textbf{prev}\ (f\ (\textbf{next}\ x)))$$

For more discussion on lift coercions in partial evaluation, including sum types, see Danvy [4].

As an example of a more realistic program in an extension of Standard ML with temporal operators, we show the regular expression matcher example from [5].

Figure 1 shows a version of the regular expression matcher without temporal operators. It makes use of a continuation function that is called with the remaining input if the current matching succeeds. The code assumes the following datatype declaration:

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

As in [5], we introduce a local function definition in the case for `acc (Star (r))` so that we can generate specialized code by applying to the first argument.

Then, we can add in temporal constructors to get the staged program in Figure 2 with the following types (using `O` here to represent $\bigcirc$)

```
val acc2 : regexp ->
             O((string list -> bool) ->
                (string list -> bool))
val accept2 : regexp ->
                O(string list -> bool)
```

This program is in fact identical to that in [5], except that here we use ` for **next** and ^ for **prev**. We can actually do better than this in $\lambda^{\bigcirc}$, by making the continuations static, and avoiding variable for variable redices, as shown in figure 3. This code effectively inlines continuations and applies them to code representing the (dynamic) strings, which will contain free variables.

5. Conclusion

We have demonstrated that the image of a small temporal logic under the Curry-Howard isomorphism, $\lambda^{\bigcirc}$, provides a logical construction of a binding-time type system that is equivalent to the core of those used in partial evaluation. We have shown that normalization in $\lambda^{\bigcirc}$ can be done in the order of the times in the logic, thus giving an explanation for why $\lambda^{\bigcirc}$ is relevant to partial evaluation.

Further, we have shown how to extend $\lambda^{\bigcirc}$ to get a small temporal functional language Mini-ML$^{\bigcirc}$ which is very similar to a realistic binding-time type system. In particular, Mini-ML$^{\bigcirc}$ allows programs that manipulate code with free variables, and we give an operational semantics which reflects this. This is in contrast to work by Davies and Pfenning [5] on Mini-ML$^{\square}$, a typed language based on modal logic that also expresses a form of binding-times, though only allows programs that manipulate closed code.

However, the manipulation of code with free variables comes at a price. Since Mini-ML$^{\bigcirc}$ does not express closed code, it can not be directly extended with a construct like that in Mini-ML$^{\square}$ that expresses immediate evaluation of generated code. Such a construct is essential in a language

```
(*  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 1. Unstaged regular expression matcher

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

Figure 2. Temporally staged regular expression matcher

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

(* val accept3 : regexp ->
                  O(string list -> bool) *)
fun accept3 r =
   ` fn s =>
     ^(acc3 r
           (fn s1 => `case ^s1 of nil => true
                                | (x::l) => false)
           `s)
```

Figure 3. Better temporally staged regular expression matcher

that supports general forms of staged computation, and is the main novel feature of Mini-ML$^{\Box}$, so in future work we will consider how to construct a type system that captures both closed code and code with free variables.

One possible direction for this work is based on the observation that manipulation of code with free variables is allowed in $\lambda^{\bigcirc}$ because there is only a single successor stage from any stage, which corresponds to the fact that $\lambda^{\bigcirc}$ is based on a linear-time temporal logic. In Mini-ML$^{\Box}$ we allow each stage to have several successor stages in order to allow more general forms of staged computation, in particular run-time code generation and sharing of code between stages (see [5] for details). This means that when constructing code in an arbitrary successor stage we cannot use variables that are bound further out in a possibly different successor stage.

This suggests that to design a language which expresses both closed code and code with free variables we could explicitly name stages and provide an explicit quantifier over them, rather than using **next** and **prev** to move between stages. This is similar to the systems of labelled natural deduction of Gabbay and de Queiroz [7], which allow many different logics to be formulated including modal logics, though this is still a speculative direction for future research.

We have implemented type checkers for the languages $\lambda^{\bigcirc}$ and $\lambda^{\mathbf{m}}$ in the logic programming language Elf (see Pfenning [16]). Using logic programming variables, the same programs will also perform type inference. We have also implemented the translations and proof of equivalence between these languages in Elf.

6. Acknowledgements

The author gratefully acknowledges discussions with Andrzej Filinski, Flemming Nielson, Jens Palsberg, and Frank Pfenning regarding the subject of this paper. The author would also like to give special thanks to Olivier Danvy for motivating and inspiring this work.

Finally, I would like to thank BRICS for offering a very stimulating and pleasant environment during my visit in the summer of 1995.

References

[1] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract types. *Science of Computer Programming*, 16:151–195, 1991.

[2] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[3] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[4] O. Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, Jan. 1996.

[5] R. Davies and F. Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 258–270, Jan. 1996.

[6] T. Freeman and F. 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.

[7] D. M. Gabbay and R. J. de Queiroz. Extending the Curry-Howard interpretation to linear, relevant and other resource logics. *Journal of Symbolic Logic*, 57:1319–1365, 1992.

[8] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, Sept. 1995.

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

[10] J. Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In S. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 1995.

[11] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1980*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard, 1979.

[12] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[13] S. Martini and A. Masini. A computational interpretation of modal proofs. In H. Wansing, editor, *Proof Theory of Modal Logics*. Kluwer, 1996. To appear.

[14] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

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

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

[17] C. Stirling. Modal and temporal logics. In S. Abramsky, D. M. Gabby, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2*, chapter 5, pages 477–563. Oxford University Press, Oxford, 1992.

[18] M. Welinder. Very efficient conversions. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *The 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications, Aspen Grove, Utah*, volume 971 of *Lecture Notes in Computer Science*, pages 340–352. Springer Verlag, September 1995.