A Modal Type System for Multi-Level Generating Extensions with Persistent Code

Yosihiro Yuse Atsushi Igarashi
Graduate School of Informatics, Kyoto University
{yuse,igarashi}@kuis.kyoto-u.ac.jp

Abstract

Multi-level generating extensions, studied by Glück and Jørgensen, are generalization of (two-level) program generators, such as parser generators, to arbitrary many levels. By this generalization, the notion of *persistent code*—a quoted code fragment that can be used for different stages—naturally arises.

In this paper we propose a typed lambda calculus $\lambda^{\bigcirc\Box}$, based on linear-time temporal logic, as a basis of programming languages for multi-level generating extensions with persistent code. The key idea of the type system is correspondence of (1) linearly ordered times in the logic to computation stages; (2) a formula $\bigcirc\, A$ (next A) to a type of code that runs at the next stage; and (3) a formula $\Box\, A$ (always A) to a type of persistent code executable at and after the current stage. After formalizing $\lambda^{\bigcirc\Box}$, we prove its key property of *time-ordered normalization* that a well-typed program can never *go back to a previous stage* in a "time-ordered" execution, as well as basic properties such as subject reduction, confluence and strong normalization. Commuting conversion plays an important role for time-ordered normalization to hold.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; F.4.1 [*Theory of Computation*]: Mathematical Logic and Formal Languages—Computational Logic, Lambda Calculus and Related Systems, Temporal Logic; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Partial Evaluation

General Terms Languages, Theory

Keywords Curry-Howard isomorphism, Meta-programming, Modal logic, Temporal logic, Time-ordered normalization, Type systems

1. Introduction

1.1 Background

Program generation and related techniques such as partial evaluation [12] have been drawing much attention as computation in a program can often be "staged" and a program specialized with respect to earlier inputs can be much faster than a general-purpose program taking all inputs at once. A good example of program generators is parser generators such as yacc, which are *two-level* program generators. Glück and Jørgensen [9] generalized two-level program generators into multi-level ones, called *multi-level generating extensions*, which, given an input, generates another program generator as an output. They showed that an ordinary program can be translated to a multi-level generating extension by exploiting multi-level binding time analysis—a natural generalization of binding-time analysis, used in off-line partial evaluation [12].

Davies [5] developed a typed λ-calculus $\lambda^{\bigcirc}$ and argued that a type system for the multi-level binding-time analysis corresponds to a proof system of (intuitionistic) linear-time temporal logic with modality "next" via the Curry-Howard isomorphism. The key idea is correspondence between the notion of time in the logic and binding-time, between a formula $\bigcirc\, A$, meaning "A holds at the next time," and a type A at the next binding-time. Moreover, term constructors **next** and **prev** to introduce and eliminate $\bigcirc\, A$, respectively, are considered quasi-quote and unquote, respectively, à la Lisp. So, terms of $\lambda^{\bigcirc}$ themselves can be considered multi-level generating extensions, which generate quoted code when executed.

Although one of the original motivations of Glück and Jørgensen's work was *automatic generation* of multi-level generating extensions, Davies and Pfenning [6, 5] and Taha and Sheard [25] argued that it would be worth studying language support for (manually written) multi-level programs. Taha et al. extended $\lambda^{\bigcirc}$ with features of run-time code generation and *persistent code*—code fragment that can be used in any future stage. Persistent code, which is useful as it enables reuse of specialized code over multiple stages, naturally arises when more than two stages are taken into account. A series of type systems [25, 18, 3, 24, 4] has been developed to ensure safety of such features.

1.2 Our Goal and Approach

Similarly to the previous work, our goal is to develop a theoretical foundation for statically typed programming languages for multi-level programs. We approach the goal by making use of the Curry-Howard isomorphism—a principle which has proven to be useful to design new type systems—as much as possible, as Davies and Pfenning [6, 5] did. More concretely, starting from $\lambda^{\bigcirc}$ (or, equivalently linear-time temporal logic with "next"), we introduce persistent code by augmenting the corresponding logic with a new modality "always". Just as a formula $\bigcirc\, A$ corresponds to a type of code of the next level, a formula $\Box\, A$ (read "always A") will correspond to a type of *persistent* code that can be used at the current or any later stages. The distinction between the two kinds of code mostly corresponds to that between open and closed code [18], whose combination has been studied already. As far as we know, however, their combination in the type system based a proof system of a temporal logic is new. Although the type system presented in this paper does not seem to subsume the previous type systems [25, 18, 3], we be-

PPDP'06 July 10–12, 2006, Venice, Italy.

lieve a logically motivated foundation will shed a new light on the design of type systems for multi-level programs.

1.3 Contributions

Our technical contributions are summarized as follows:

- We develop a typed calculus $\lambda^{\bigcirc\Box}$, corresponding to linear-time temporal logic with two modalities "next" and "always" and give its formal definition consisting of syntax, a type system, and operational semantics;
- We prove that $\lambda^{\bigcirc\Box}$ enjoys basic properties such as subject reduction, confluence, strong normalization; and
- We also prove the property of *time-ordered normalization*, first formulated and proved by Davies [5] for $\lambda^{\bigcirc}$.

Intuitively, time-ordered normalization states that, in staged execution of a well-typed program, earlier stage execution never requires that of future stages. So, this property can be considered one of the most important correctness criteria of type systems for multi-level programs, as well as standard ones such as type safety.

Davies and Pfenning [6] studied the Curry-Howard isomorphism for modal logic S4 (with necessity) by developing another typed λ-calculus $\lambda^{\Box}$, and discussed its connection with run-time code generation. Our calculus $\lambda^{\bigcirc\Box}$ is not a straightforward combination of two calculi $\lambda^{\bigcirc}$ and $\lambda^{\Box}$—it turns out that non-trivial extensions are required for time-ordered normalization, a crucial property as a calculus for multi-level programs, to hold.

1.4 The Rest of This Paper

First, we show an overview of the calculus $\lambda^{\bigcirc\Box}$ with programming examples in Section 2. We give its formal definition in Section 3 with proofs of its properties including time-ordered normalization; then, we develop Mini-ML$^{\bigcirc\Box}$ by extending $\lambda^{\bigcirc\Box}$ with basic data types and recursion and give its call-by-value semantics in Section 4. Finally, after discussing related work in Section 5, we conclude in Sect. 6. We omit most of proofs here; a full version is available at `http//www.sato.kuis.kyoto-u.ac.jp/~yuse/papers/`.

2. Overview of $\lambda^{\bigcirc\Box}$

In this section, we first describe how a proof system of linear-time temporal logic can evolve to a typed calculus $\lambda^{\bigcirc\Box}$ and then explain how the term constructors related to the modalities can be considered as program constructs that manipulate code expressions. The key ideas for relating this kind of logic with a calculus are the following three correspondences:

- the linearly ordered times in the logic to computation stages in the calculus,
- formula $\bigcirc A$ to the type of *ephemeral code* (of type A), that is, code which will be executed only at the next stage, and
- formula $\Box A$ to the type of *persistent code* (of type A), which can be executed at the current and any later stage.

Finally, we informally discuss time-ordered normalization in $\lambda^{\bigcirc\Box}$ with some technical subtleties.

2.1 A Proof System of Linear-time Temporal Logic

We first consider a proof system for linear-time temporal logic with two modalities—$\bigcirc A$, which means "A at the next time," and $\Box A$, which means "always A from now on." The proof system is partially inspired by Pfenning and Davies's formalization of modal logic [20], based on the notion of judgments [15]. The basic idea is to consider two kinds of judgments "A holds at time n" and "A holds at time n and any later." Accordingly, a hypothetical judgment has two kinds of assumption sets and takes the following form:

$$A_1^{n_1}, \ldots, A_k^{n_k}; B_1^{m_1}, \ldots, B_\ell^{m_\ell} \vdash^n C$$

which means "C holds at time n under the assumption that each A_i holds at time n_i and any later and that each B_j holds (only) at time m_j." Then, the former kind of assumptions can be used only when the current time is later than their time, while the latter can be used only when the current time and their time agree, resulting in the following two rules:

$$\frac{n_i \leq m}{\ldots, A_i^{n_i}, \ldots; \Gamma \vdash^m A_i} \qquad \frac{}{\Delta; \ldots, B_i^m, \ldots \vdash^m B_i}$$

Usual connectives are formulated in a standard manner, except the form of hypothetical judgments. For example, the introduction and elimination rules for implication are as follows:

$$\frac{\Delta; \Gamma, A^n \vdash^n B}{\Delta; \Gamma \vdash^n A \to B} \qquad \frac{\Delta; \Gamma \vdash^n A \to B \qquad \Delta; \Gamma \vdash^n A}{\Delta; \Gamma \vdash^n B}$$

Note that the times attached to judgments have to agree.

Considering the intuitive meaning of a hypothetical judgment, we can easily give the introduction and elimination rules for modality $\bigcirc$:

$$\frac{\Delta; \Gamma \vdash^{n+1} A}{\Delta; \Gamma \vdash^n \bigcirc A} \qquad \frac{\Delta; \Gamma \vdash^n \bigcirc A}{\Delta; \Gamma \vdash^{n+1} A}$$

Modality $\Box$ is expressed by a hypothetical judgment with zero assumptions of the form "A holds at time n". The introduction rule for modality $\Box$ amounts to internalizing such a (hypothetical) judgment, and the elimination rule turns "$\Box A$ holds at time n" into "A holds at time n and any later":

$$\frac{\Delta; \cdot \vdash^n A}{\Delta; \Gamma \vdash^n \Box A} \qquad \frac{\Delta; \Gamma \vdash^{n+i} \Box A \qquad \Delta, A^{n+i}; \Gamma \vdash^n B}{\Delta; \Gamma \vdash^n B}$$

(Here, $\cdot$ denotes the empty assumption set.) We allow the elimination of $\Box A$ at a time different from the conclusion for reasons discussed later.

2.2 Proof Terms for Code Manipulation

By the Curry-Howard isomorphism, judgments of a proof system corresponds to type judgments of a calculus, by augmenting with variables and terms. Since the hypothetical judgment form include two assumption sets, a type judgment naturally takes the following form:

$$u_1 ::^{n_1} A_1, \ldots, u_k ::^{n_k} A_k; x_1 :^{m_1} B_1, \ldots, x_\ell :^{m_\ell} B_\ell \vdash^n M : C$$

with two kinds of variables u_i and x_j. We call the former *persistent variables* as they are bound to persistent code and the latter *ordinary variables*. Intuitively, the type judgment means that term M is given type C at time n under the condition that each persistent variable u_i of type A_i can be used at time n_i or later, and each ordinary variable x_j has type B_j at m_j.

Proof rules in a logic correspond to typing rules in a calculus. So, we obtain the following rules from introduction/elimination rules for $\bigcirc$:

$$\frac{\Delta; \Gamma \vdash^{n+1} M : A}{\Delta; \Gamma \vdash^n \mathbf{next}\, M : \bigcirc A} \qquad \frac{\Delta; \Gamma \vdash^n M : \bigcirc A}{\Delta; \Gamma \vdash^{n+1} \mathbf{prev}\, M : A}\ .$$

Since proof steps in which an introduction rule is followed by an elimination rule correspond to a redex, we also obtain the reduction rule $\mathbf{prev}(\mathbf{next}\, M) \longrightarrow M$. From a computational point of view, **next** and **prev** are considered Lisp's quasi-quotation (‘) and

unquote (`,`), respectively. Thus, the following reduction sequence

$$\begin{aligned}&\underline{(\lambda z.\ \mathbf{next}(x + \mathbf{prev}\ z))\ (\mathbf{next}\ 1)}\\ &\longrightarrow_\beta \mathbf{next}(x + \underline{\mathbf{prev}\ \mathbf{next}\ 1})\\ &\longrightarrow_\bigcirc \mathbf{next}(x+1)\end{aligned}$$

can be seen as the evaluation of `((lambda (z) '(+ x ,z)) '1)` to `'(+ x 1)`.

Similarly, from the proof rules for $\Box A$, we obtain

$$\frac{\Delta;\cdot \vdash^n M : A}{\Delta;\Gamma \vdash^n \mathbf{box}\, M : \Box A}$$

$$\frac{\Delta;\Gamma \vdash^{n+i} M : \Box A \qquad \Delta, u{::}^{n+i}A;\Gamma \vdash^n N : B}{\Delta;\Gamma \vdash^n \mathbf{let\ box}\, u =_i M\ \mathbf{in}\ N : B}$$

as typing rules and $\mathbf{let\ box}\, u =_i \mathbf{box}\, M\ \mathbf{in}\ N \longrightarrow [M/u]N$ as a reduction rule. (Here, $[M/u]$ is substitution of M for a persistent variable u.) From a computational viewpoint, $\mathbf{box}\, M$ constructs persistent code. Since persistent code may be executed at different stages, M should not contain free ordinary variables, which are available only at a certain stage. Roughly speaking, $\mathbf{let\ box}\, u =_i M\ \mathbf{in}\ N$ first evaluates M to obtain persistent code $\mathbf{box}\, M'$, binds u to M' by removing **box**, and evaluates N. For example, the following reduction sequence demonstrates how persistent code can be useful:

$$\begin{aligned}&\underline{\mathbf{let\ box}\, u =_0 \mathbf{box}(\lambda x.\ x+1)\ \mathbf{in}\ (u\ 2, \mathbf{next}(u\ 3, \mathbf{next}\ u\ 4))}\\ &\longrightarrow_\Box ((\underline{\lambda x.\ x+1)\ 2}, \mathbf{next}((\lambda x.\ x+1)\ 3, \mathbf{next}(\lambda x.\ x+1)\ 4))\\ &\longrightarrow_\beta (2+1, \mathbf{next}((\lambda x.\ x+1)\ 3, \mathbf{next}(\lambda x.\ x+1)\ 4))\end{aligned}$$

Notice that the persistent code $\lambda x.\ x+1$ is embedded into the code of different stages; furthermore, we can view embedding persistent code into the current stage (as in $u\ 2$) as run-time code generation, because it runs at the same stage as it is constructed.

2.3 Commuting Conversions for Time-Ordered Normalization

Since a $\lambda^{\bigcirc\Box}$ term is a staged program, the reduction relation is augmented with information on *when* or *at which stage* reduction occurs: for example, $(\lambda x.\ x)y \xrightarrow{0}_\beta y$ while $\mathbf{next}((\lambda x.\ x)y) \xrightarrow{1}_\beta \mathbf{next}\, y$, as execution inside quotation **next** happens at the next stage. Time-ordered normalization roughly states that a normal form of a given well-typed term can be obtained by reducing in the (increasing) order of stages. In other words, after reduction at a certain stage finishes, there will be no need to go back to this stage. For example, the reduction stages of the following sequence is in the increasing order of stages:

$$\begin{aligned}&\underline{(\lambda x.\ \mathbf{next}((\mathbf{prev}\ x)\ y))\ \mathbf{next}\ \lambda z.\ z}\\ &\xrightarrow{0}_\beta \mathbf{next}((\underline{\mathbf{prev}\ \mathbf{next}\ \lambda z.\ z})\ y)\\ &\xrightarrow{0}_\bigcirc \mathbf{next}(\underline{(\lambda z.\ z)\ y})\\ &\xrightarrow{1}_\beta \mathbf{next}\ y\end{aligned}$$

This property is thus an evidence that the type system correctly captures computation stages.

Unfortunately, the reduction rules described above are not sufficient for time ordered normalization to hold:

$$\begin{aligned}&(\mathbf{let\ box}\, u =_1 \underline{(\lambda x.\ x)\ \mathbf{box}\, v}\ \mathbf{in}\ \lambda y.\ y)\ z\\ &\xrightarrow{1}_\beta \underline{(\mathbf{let\ box}\, u =_1 \mathbf{box}\, v\ \mathbf{in}\ \lambda y.\ y)}\ z\\ &\xrightarrow{1}_\Box \underline{(\lambda y.\ y)\ z}\\ &\xrightarrow{0}_\beta z\end{aligned}$$

Here, the first reduction belongs to the stage 1 since **let box** tries to destruct persistent code of the next stage (notice that $=_1$). The problem is that **let box** for a future stage blocks a β-redex of the current stage.

In order to solve this problem, we adopt commuting conversions [8], which are often found in calculi with sum types and case expressions:

$$(\mathbf{let\ box}\, u =_i P\ \mathbf{in}\ Q)\ R \xrightarrow{0}_{\mathrm{com}} \mathbf{let\ box}\, u =_i P\ \mathbf{in}\ (Q\ R)$$

This rule, which expands the scope of u of **let box**, may reveal a hidden redex $Q\ R$ (when Q is a λ-abstraction). Then, the problem above disappears as follows:

$$\begin{aligned}&\underline{(\mathbf{let\ box}\, u =_1 (\lambda x.\ x)\ \mathbf{box}\, v\ \mathbf{in}\ \lambda y.\ y)\ z}\\ &\xrightarrow{0}_{\mathrm{com}} \mathbf{let\ box}\, u =_1 (\lambda x.\ x)\ \mathbf{box}\, v\ \mathbf{in}\ \underline{(\lambda y.\ y)\ z}\\ &\xrightarrow{0}_\beta \mathbf{let\ box}\, u =_1 \underline{(\lambda x.\ x)\ \mathbf{box}\, v}\ \mathbf{in}\ z\\ &\xrightarrow{1}_\beta \underline{\mathbf{let\ box}\, u =_1 \mathbf{box}\, v\ \mathbf{in}\ z}\\ &\xrightarrow{1}_\Box z\ .\end{aligned}$$

Similar observations have been made by Lawall and Thiemann [14] and Hatcliff and Davies [11], who used (an extension of) Moggi's computational lambda-calculus [17] as a formal framework of partial evaluation.

2.4 Programming in Mini-ML$^{\bigcirc\Box}$

Now, we extend $\lambda^{\bigcirc\Box}$ with familiar programming constructs such as the Boolean type and recusive definitions to Mini-ML$^{\bigcirc\Box}$and demonstrate simple examples of programming to compare different modalities. As in Davies and Pfenning [5, 6], we take the standard example of the recursively defined power function x^n, and compare several programs to generate specialized code with respect to the exponent n given as an input. Although the presentation here is rather informal—for example, we often omit obvious type declarations in favor of readability—we will give the definition of Mini-ML$^{\bigcirc\Box}$ in Section 4 with its call-by-value reduction semantics.

Taking the meaning of **next** and **box** as quotation into account, we assume that a term inside a quotation never reduces unless it is escaped with **prev**.

We begin with an ordinary definition of `power` without staging (here, $\rightsquigarrow^*$ is the call-by-value reduction relation defined later):

$$\begin{aligned}&\texttt{power} : \mathbf{int} \to \mathbf{int} \to \mathbf{int}\\ &\quad \equiv \mathbf{fix}\, p{:}\mathbf{int}\to\mathbf{int}\to\mathbf{int}.\ \lambda n{:}\mathbf{int}.\\ &\qquad \mathbf{if}\ n = 0\ \mathbf{then}\ \lambda x{:}\mathbf{int}.\ 1\\ &\qquad\qquad \mathbf{else\ let}\ q = p\ (n-1)\ \mathbf{in}\ \lambda x{:}\mathbf{int}.\ x * (q\ x)\\ &\texttt{power}\ 2 \rightsquigarrow^*\ \lambda x.\ x * ((\lambda x.\ x * ((\lambda x.\ 1)\ x))\ x)\end{aligned}$$

Note that, under a usual language implementation, the result of `power` 2 is a function value, which does not have a symbolic representation. Now, we augment the function body above with **next**, **prev**, **box** and **let box** to obtain functions, which return a specialized program as quoted code.

First, we show `power_b`, which uses only **box** and **let box** as in Davies and Pfenning [6]:

$$\begin{aligned}&\texttt{power\_b} : \mathbf{int} \to \Box(\mathbf{int} \to \mathbf{int})\\ &\quad \equiv \mathbf{fix}\, p{:}\mathbf{int}\to\Box(\mathbf{int}\to\mathbf{int}).\\ &\qquad \lambda n{:}\mathbf{int}.\ \mathbf{if}\ n = 0\ \mathbf{then}\ \mathbf{box}(\lambda x{:}\mathbf{int}.\ 1)\\ &\qquad\qquad \mathbf{else\ let\ box}\, u = p\ (n-1)\\ &\qquad\qquad\quad \mathbf{in}\ \mathbf{box}(\lambda x{:}\mathbf{int}.\ x * (u\ x))\\ &\texttt{power\_b}\ 2 \rightsquigarrow^*\ \mathbf{box}(\lambda x.\ x * ((\lambda x.\ x * ((\lambda x.\ 1)\ x))\ x))\end{aligned}$$

The residual code generated by `power_b`, however, still contains redundant redices, like $(\lambda x.\, 1)\ x$, and thus is not such an efficient one. (Note that reduction under **box** does not occur.)

Davies [5] showed that those redundant redices can be eliminated with $\lambda^{\bigcirc}$, thanks to the ability to manipulate code that contains free variables:

$$\begin{aligned}
&\texttt{power_c} : \mathsf{int} \to \bigcirc(\mathsf{int} \to \mathsf{int}) \\
&\quad \equiv \lambda n{:}\mathsf{int}.\ \mathsf{next}(\lambda x{:}\mathsf{int}.\ \mathsf{prev}(\\
&\qquad (\mathsf{fix}\, p{:}\mathsf{int}{\to}\bigcirc\mathsf{int}.\ \lambda m{:}\mathsf{int}. \\
&\qquad\quad \mathsf{if}\ m = 0\ \mathsf{then}\ \mathsf{next}\, 1 \\
&\qquad\qquad\qquad \mathsf{else}\ \ \mathsf{next}(x * \mathsf{prev}(p\ (m-1)))) \\
&\qquad n)) \\
&\texttt{power_c}\ 2 \rightsquigarrow^* \mathsf{next}(\lambda x.\, x * (x * 1))
\end{aligned}$$

The key idea in this definition is to use **prev** inside the scope of λ-abstracted variable x, compute code of the form $\mathsf{next}(\underbrace{x * x * \cdots * x}_{n} * 1)$ of type $\bigcirc\,\mathsf{int}$, and embed it into the body of the specialized function. Notice that **fix** is moved deep inside of the whole function body. `power_c` provides more efficient code than `power_b`, but the code itself is restricted in reuse: it can be executed only at the next stage. Thus, if one wants to use specialized power functions two stages later, he or she also has to write another, very similar version with type $\mathsf{int} \to \bigcirc\bigcirc(\mathsf{int} \to \mathsf{int})$.

One of our goals is to combine the flexibility of **next** code and portability of **box** code. Now that our calculus allows both $\bigcirc$ and $\Box$ in one program, we can define new function `power_cb` as follows:

$$\begin{aligned}
&\texttt{power_cb} : \Box\mathsf{int} \to \bigcirc\,\Box(\mathsf{int} \to \mathsf{int}) \\
&\quad \equiv \lambda n'{:}\Box\mathsf{int}.\ \mathsf{let}\ \mathsf{box}\, u = n'\ \mathsf{in}\ \mathsf{next}\,\mathsf{box}\,\lambda x{:}\mathsf{int}.\ \mathsf{prev}(\\
&\qquad (\mathsf{fix}\, p{:}\mathsf{int}{\to}\bigcirc\mathsf{int}.\ \lambda m{:}\mathsf{int}. \\
&\qquad\quad \mathsf{if}\ m = 0\ \mathsf{then}\ \mathsf{next}\, 1 \\
&\qquad\qquad\qquad \mathsf{else}\ \ \mathsf{next}(x * \mathsf{prev}(p\ (m-1)))) \\
&\qquad u) \\
&\texttt{power_cb}\ (\mathsf{box}\, 2) \rightsquigarrow^* \mathsf{next}\,\mathsf{box}(\lambda x.\, x * (x * 1))
\end{aligned}$$

As seen above, the resulting code is as efficient as in `power_c`. Moreover, the code has type $\bigcirc\,\Box(\mathsf{int} \to \mathsf{int})$ and so can be used at the next and any later stage: for example, the next term shows that a specialized cube function is used at multiple stages.

$$\begin{aligned}
&\mathsf{let}\ \mathsf{box}\, u =_1 \mathsf{prev}(\texttt{power_cb}(\mathsf{box}\, 3)) \\
&\mathsf{in}\ \mathsf{next}(\cdots u\, 5 \cdots \mathsf{next}(\cdots u\, 9 \cdots) \cdots)
\end{aligned}$$

The type of this function, however, is not quite what may have been expected: it takes $\Box\,\mathsf{int}$, not **int**, as an argument and returns $\bigcirc\,\Box(\mathsf{int} \to \mathsf{int})$, not $\Box(\mathsf{int} \to \mathsf{int})$, which could be used even at the current stage by run-time code generation. The first problem stems from the fact that, unlike `power_b`, the argument has to be used inside **box**, due to the move of **fix**. We do not think this is a significant problem because constants of base types (such as 2, **true** etc.) are actually available at any stage and so we can assume that they always have an box type (such as $\Box\,\mathsf{int}$, $\Box\,\mathsf{bool}$ etc.). The second problem in the return type from the fact that the only way to escape from quotation is to use **prev**, which refers to the previous stage. If we omitted **next** before **box**, the variable u could not be referred to inside **prev**, because **box** does not change the stage at which a term is typed. It is left for future work to solve this problem with a type system that corresponds to logic via the Curry-Howard isomorphism.

3. $\lambda^{\bigcirc\Box}$

In this section, we give a formal definition of $\lambda^{\bigcirc\Box}$ with its syntax (Section 3.1), type system (Section 3.2), and operational semantics (Section 3.3). Finally, we discuss some properties of $\lambda^{\bigcirc\Box}$ in Section 3.4.

3.1 Syntax

Let $\mathbf{OVars}$ and $\mathbf{PVars}$ be countably infinite sets of *ordinary variables*, ranged over by x and y, and *persistent variables*, ranged over by u and v, respectively. We assume $\mathbf{OVars}$ and $\mathbf{PVars}$ are disjoint. We also assume the set $\mathbf{BTypes}$ of *base types*, ranged over by b.

DEFINITION 1 (Types and Terms). *The sets of* types, *ranged over by A and B, and* terms, *ranged over by M and N, are defined by the following grammar:*

$$\begin{aligned}
&\textit{types} \quad A, B ::= b \mid A \to B \mid \bigcirc A \mid \Box A \\
&\textit{terms} \quad M, N ::= x \mid u \mid \lambda x{:}A.\ M \mid M\ N \mid \mathsf{next}\, M \mid \mathsf{prev}\, M \\
&\qquad\qquad \mid \mathsf{box}\, M \mid \mathsf{let}\ \mathsf{box}\, u =_i M\ \mathsf{in}\ N
\end{aligned}$$

where i is a natural number.

The persistent variable u of $\mathsf{let}\ \mathsf{box}\, u =_i N\ \mathsf{in}\ M$ and the ordinary variable x of $\lambda x{:}A.\ M$ are bound in M. In what follows, we assume tacit renaming of bound variables so that no bound variable has the same name as any other bound variable or free variable. The type constructors $\bigcirc$ and $\Box$ connect tighter than $\to$, so, for example, $\bigcirc\, A \to B$ means $(\bigcirc\, A) \to B$. **let box** and λ extends to the right as much as possible: for example, $\mathsf{let}\ \mathsf{box}\, u =_i M\ \mathsf{in}\ x\, y$ means $\mathsf{let}\ \mathsf{box}\, u =_i M\ \mathsf{in}\ (x\ y)$. The index i in **let box** is often omitted when it is 0.

We write $\mathrm{FPV}(M)$ for the set of free persistent variables and $\mathrm{FOV}(M)$ for the set of free ordinary variables, both of which are defined in the standard manner. We write $[M/x]$ and $[M/u]$ for the standard capture-avoiding substitution of M for x and u, respectively.

3.2 Type System

As discussed in the previous section, the form of the type judgments is $\Delta; \Gamma \vdash^n M : A$, read "term M is given type A at time (stage) n under persistent context Δ and ordinary context Γ." Here, n is a natural number. Persistent and ordinary contexts are formally defined as follows:

$$\begin{array}{lrcl}
\text{persistent context} & \Delta & ::= & \cdot \mid \Delta, u{::}^n A \\
\text{ordinary context} & \Gamma & ::= & \cdot \mid \Gamma, x{:}^n A
\end{array}$$

We assume that no variables are declared more than once in one context.

The typing rules to derive type judgments are shown in Figure 1. Each rule is obtained by augmenting each inference rule in the previous section with term constructors. The rules T-OVAR and T-PVAR correspond to the proof rules for truth and valid assumptions, respectively. Notice that a declaration $u{::}^n A$ in a persistent context can be referred to at any time equal to or later than n. The rules T-OVAR, T-ABS, and T-APP for variables, abstractions, and applications, respectively, are essentially the same as ones in the simply typed λ calculus. Notice that the times of type judgments in the premises and conclusions must be the same.

An intuitive meaning of the other rules, related to code types $\bigcirc\, A$ and $\Box\, A$, are explained in the previous section. As already mentioned, T-LETBOX allows to unquote persistent code M of time $n + i$ ($i \geq 0$), which can be later than the time of the body N. Allowing $i > 0$ seems necessary to "prove" both $\bigcirc\,\Box\, A \to \Box\,\bigcirc\, A$ and $\Box\,\bigcirc\, A \to \bigcirc\,\Box\, A$ for any A. The next example shows such usage of T-LETBOX.

EXAMPLE 1 ($\Box\,\bigcirc\, A \leftrightarrow \bigcirc\,\Box\, A$).
Let $M_1 \equiv \lambda x{:}\Box\,\bigcirc\, A.\ \mathsf{let}\ \mathsf{box}\, u =_0 x\ \mathsf{in}\ \mathsf{next}\,\mathsf{box}\,\mathsf{prev}\, u$ *and* $M_2 \equiv \lambda x{:}\bigcirc\,\Box\, A.\ \mathsf{let}\ \mathsf{box}\, u =_1 \mathsf{prev}\, x\ \mathsf{in}\ \mathsf{box}\,\mathsf{next}\, u$. *Then, we*

$$\frac{(x{:}^n A \in \Gamma)}{\Delta; \Gamma \vdash^n x : A} \quad \text{(T-OVAR)}$$

$$\frac{(u{::}^m A \in \Delta \qquad n \geq m)}{\Delta; \Gamma \vdash^n u : A} \quad \text{(T-PVAR)}$$

$$\frac{\Delta; \Gamma, x{:}^n A \vdash^n M : B}{\Delta; \Gamma \vdash^n \lambda x{:}A.\, M : A \to B} \quad \text{(T-ABS)}$$

$$\frac{\Delta; \Gamma \vdash^n M : A \to B \qquad \Delta; \Gamma \vdash^n N : A}{\Delta; \Gamma \vdash^n M\, N : B} \quad \text{(T-APP)}$$

$$\frac{\Delta; \Gamma \vdash^{n+1} M : A}{\Delta; \Gamma \vdash^n \mathbf{next}\, M : \bigcirc A} \quad \text{(T-NEXT)}$$

$$\frac{\Delta; \Gamma \vdash^n M : \bigcirc A}{\Delta; \Gamma \vdash^{n+1} \mathbf{prev}\, M : A} \quad \text{(T-PREV)}$$

$$\frac{\Delta; \cdot \vdash^n M : A}{\Delta; \Gamma \vdash^n \mathbf{box}\, M : \Box A} \quad \text{(T-BOX)}$$

$$\frac{\Delta; \Gamma \vdash^{n+i} M : \Box A \qquad \Delta, u{::}^{n+i} A; \Gamma \vdash^n N : B \qquad (i \geq 0)}{\Delta; \Gamma \vdash^n \mathbf{let\ box}\, u =_i M\, \mathbf{in}\, N : B} \quad \text{(T-LETBOX)}$$

Figure 1. Typing rules

have typing derivations $\cdot; \cdot \vdash^0 M_1 : \Box \bigcirc A \to \bigcirc \Box A$ *and* $\cdot; \cdot \vdash^0 M_2 : \bigcirc \Box A \to \Box \bigcirc A$ *as shown in Figure 2. Note that in the latter case the* T-LETBOX *rule allows the box code to be used at a time different from that of the code itself.*

Closed terms of these types, together with those of types $\Box A \to A$ and $\Box A \to \Box \Box A$ (they are axioms of S4 modal logic) for any A, make it possible to convert any code type $\bigstar_1 \cdots \bigstar_n A$, where $\bigstar_i$ stands for either $\bigcirc$ or $\Box$, to a "normalized" code type $\bigcirc \cdots \bigcirc \Box A$ or $\bigcirc \cdots \bigcirc A$—which concisely expresses at which time code is available—and back.

3.3 Operational Semantics

We define the operational semantics of $\lambda^{\bigcirc\Box}$ with the (ternary) reduction relation $M \xrightarrow{k} M'$, read "term M reduces to M' in one step k stages later." The index k can be negative. The reduction rules are shown in Figure 3. We simply write $M \longrightarrow M'$ if $M \xrightarrow{k} M'$ for some k and $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

The rule R-BETA is for the standard β-reduction. The R-PREV is for unquoting of (ephemeral) code expression. Note that the index is -1 since the redex is typically under another quotation of the previous stage and unquoting happens during the execution of the previous stage. The rule R-LETBOX is for embedding persistent code expression; if u appears not under **box**, the quoted code N will run during the execution of the current stage. The rules R-APPC, R-PREVC, and LETBOXC are for commuting conversions, we mentioned in the previous section. We provide congruence rules, RC-ABS etc., for each term constructors, as we assume a full reduction system here. Note that some term constructors change the time of the term, so the index k should be changed accordingly.

EXAMPLE 2. *Let* $M \equiv \mathbf{prev}(\mathbf{let\ box}\, u =_2 \mathbf{box}\, P\, \mathbf{in}\, Q)$. *Then,* $M \xrightarrow{1} \mathbf{prev}([P/u]Q)$ *by* R-LETBOX, RC-PREV *and* $M \xrightarrow{-1} \mathbf{let\ box}\, u =_1 \mathbf{box}\, P\, \mathbf{in\ prev}\, Q$ *by* R-PREVC.

3.4 Properties of $\lambda^{\bigcirc\Box}$

The calculus $\lambda^{\bigcirc\Box}$ enjoys basic properties such as *subject reduction*, *confluence* and *strong normalization*, which are proved in a fairly standard manner.

THEOREM 1 (Subject Reduction). *If* $\Delta; \Gamma \vdash^n M : A$ *and* $M \longrightarrow M'$, *then* $\Delta; \Gamma \vdash^n M' : A$.

THEOREM 2 (Confluence). *For any term* M, *if* $M \longrightarrow^* M_1$ *and* $M \longrightarrow^* M_2$, *then there exists some term* N *such that* $M_1 \longrightarrow^* N$ *and* $M_2 \longrightarrow^* N$.

THEOREM 3 (Strong Normalization). *If* $\Delta; \Gamma \vdash^n M : A$, *then there exists no infinite reduction sequence such that* $M \longrightarrow M_1 \longrightarrow \cdots \longrightarrow M_n \longrightarrow \cdots$.

Now, we give a rigorous formalization and proof for *time-ordered normalization*. The intuitive meaning of time-ordered normalization is that, in staged execution of a well-typed program, earlier stage execution never requires that of future stages. In other words, having finished its execution at stage n, program execution never *goes back to this finished stage*. This property is stronger than the property such as "suitably defined call-by-value evaluation can realizes staged computation," which we will prove in the next section, because, here, a reduction strategy within a stage is arbitrary (as long as normal forms can be obtained with it).

Time-ordered normalization is first introduced by Davies and proved for $\lambda^{\bigcirc}$ [5], which is a subset of $\lambda^{\bigcirc\Box}$. His formulation (statement and proof), however, was somewhat informal and even limited to β-reduction: he showed that the time of β-redices can be ordered, assuming R-PREV is applied to all redices of the form $\mathbf{prev\ next}\, M$ between two β-reductions. In contrast, we state that property in a more formal and general manner, taking all kinds of reduction including R-BETA, R-PREV, and R-LETBOX and even commuting conversions.

To make a formal statement, we introduce the relation $\Downarrow^n M$, read "term M is a normal form w.r.t. the times less than n," together with an auxiliary judgment $\bigtriangledown^n M$, read "term M is neutral w.r.t. the times less than n," which are defined by rules in Figure 4. $\Downarrow^n M$ means that there exists no M' such that $M \xrightarrow{m} M'$ for any $m < n$, M' and, $\bigtriangledown^n M$ that sustituting M for a variable in any (well-typed) term does not yield a new redex at an earlier time. Note that **let box** is *not* neutral, since substituting **let box** for x in, say, $x\, N$ yields a new redex due to commuting conversion.

Then, the time-ordered normalization theorem can be stated as below. We write $M \xrightarrow{k}\!\!\!\!\!/\ $ if there exists no M' such that $M \xrightarrow{k} M'$.

THEOREM 4 (Time-Ordered Normalization). *If* $\Delta; \Gamma \vdash^n M : A$ *and* $\Downarrow^k M$, *then for any series of reductions at time* k *such that* $M \xrightarrow{k} M' \xrightarrow{k} \cdots \xrightarrow{k} N \xrightarrow{k}\!\!\!\!\!/$, *it holds that* $\Downarrow^{k+1} N$.

$$\dfrac{\dfrac{}{\cdot;\Gamma_1 \vdash^0 x : \Box \bigcirc A} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Delta_1;\cdot \vdash^0 u : \bigcirc A}}{\Delta_1;\cdot \vdash^1 \textsf{prev}\, u : A}}{\Delta_1;\Gamma_1 \vdash^1 \textsf{box prev}\, u : \Box A}}{\Delta_1;\Gamma_1 \vdash^0 \textsf{next box prev}\, u : \bigcirc \Box A}}{\dfrac{\cdot;\Gamma_1 \vdash^0 \textsf{let box}\, u =_0 x \,\textsf{in next box prev}\, u : \bigcirc \Box A}{\cdot;\cdot \vdash^0 M_1 : \Box \bigcirc A \to \bigcirc \Box A}}$$

$M_1 \equiv \lambda x{:}\Box \bigcirc A.\, \textsf{let box}\, u =_0 x \,\textsf{in next box prev}\, u$
$\Delta_1 \equiv u{::}^0 \bigcirc A$
$\Gamma_1 \equiv x{:}^0 \Box \bigcirc A$

$$\dfrac{\dfrac{}{\cdot;\Gamma_2 \vdash^0 x : \bigcirc \Box A} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Delta_2;\cdot \vdash^1 u : A}}{\Delta_2;\cdot \vdash^0 \textsf{next}\, u : \bigcirc A}}{\Delta_2;\Gamma_2 \vdash^0 \textsf{box next}\, u : \Box \bigcirc A}}{\dfrac{\cdot;\Gamma_2 \vdash^0 \textsf{let box}\, u =_1 \textsf{prev}\, x \,\textsf{in box next}\, u : \Box \bigcirc A}{\cdot;\cdot \vdash^0 M_2 : \bigcirc \Box A \to \Box \bigcirc A}}$$

$M_2 \equiv \lambda x{:}\bigcirc \Box A.\, \textsf{let box}\, u =_1 \textsf{prev}\, x \,\textsf{in box next}\, u$
$\Delta_2 \equiv u{::}^1 A$
$\Gamma_2 \equiv x{:}^0 \bigcirc \Box A$

Figure 2. Examples of type derivations

$$(\lambda x{:}A.\, M)\, N \xrightarrow{0} [N/x]M \qquad \text{(R-BETA)}$$

$$\textsf{prev}(\textsf{next}\, M) \xrightarrow{-1} M \qquad \text{(R-PREV)}$$

$$\textsf{let box}\, u =_i \textsf{box}\, M \,\textsf{in}\, N \xrightarrow{i} [M/u]N \qquad \text{(R-LETBOX)}$$

$$\dfrac{(i > 0 \text{ and } u \notin \mathrm{FPV}(N))}{(\textsf{let box}\, u =_i L \,\textsf{in}\, M)\, N \xrightarrow{0} \textsf{let box}\, u =_i L \,\textsf{in}\, (M\, N)} \qquad \text{(R-APPC)}$$

$$\textsf{prev}(\textsf{let box}\, u =_{i+1} L \,\textsf{in}\, M) \xrightarrow{-1} \textsf{let box}\, u =_i L \,\textsf{in}\, (\textsf{prev}\, M) \qquad \text{(R-PREVC)}$$

$$\dfrac{(j > 0 \text{ and } u \notin \mathrm{FPV}(N))}{\textsf{let box}\, t =_i (\textsf{let box}\, u =_j L \,\textsf{in}\, M) \,\textsf{in}\, N \xrightarrow{i} \textsf{let box}\, u =_{i+j} L \,\textsf{in}\, (\textsf{let box}\, t =_i M \,\textsf{in}\, N)} \qquad \text{(R-LETBOXC)}$$

$$\dfrac{M \xrightarrow{k} M'}{\lambda x{:}A.\, M \xrightarrow{k} \lambda x{:}A.\, M'} \qquad \text{(RC-ABS)}$$

$$\dfrac{M \xrightarrow{k} M'}{M\, N \xrightarrow{k} M'\, N} \qquad \text{(RC-APP1)}$$

$$\dfrac{N \xrightarrow{k} N'}{M\, N \xrightarrow{k} M\, N'} \qquad \text{(RC-APP2)}$$

$$\dfrac{M \xrightarrow{k} M'}{\textsf{next}\, M \xrightarrow{k+1} \textsf{next}\, M'} \qquad \text{(RC-NEXT)}$$

$$\dfrac{M \xrightarrow{k} M'}{\textsf{prev}\, M \xrightarrow{k-1} \textsf{prev}\, M'} \qquad \text{(RC-PREV)}$$

$$\dfrac{M \xrightarrow{k} M'}{\textsf{box}\, M \xrightarrow{k} \textsf{box}\, M'} \qquad \text{(RC-BOX)}$$

$$\dfrac{M \xrightarrow{k} M'}{\textsf{let box}\, u =_i M \,\textsf{in}\, N \xrightarrow{k+i} \textsf{let box}\, u =_i M' \,\textsf{in}\, N} \qquad \text{(RC-LETBOX1)}$$

$$\dfrac{N \xrightarrow{k} N'}{\textsf{let box}\, u =_i M \,\textsf{in}\, N \xrightarrow{k} \textsf{let box}\, u =_i M \,\textsf{in}\, N'} \qquad \text{(RC-LETBOX2)}$$

Figure 3. Reduction rules

Neutral terms:

$$\frac{(k \leq 0)}{\bigtriangledown^k M} \quad \text{(N-NEUT0)} \qquad \frac{(M\text{: } x,\ u,\ P\ Q,\ \mathbf{prev}\ P)}{\bigtriangledown^k M} \quad \text{(N-NEUT1)}$$

Normal forms:

$$\Downarrow^k x \quad \text{(N-OVAR)} \qquad \frac{\Downarrow^{k+1} P \qquad \bigtriangledown^{k+1} P}{\Downarrow^k (\mathbf{prev}\ P)} \quad \text{(N-PREV)}$$

$$\Downarrow^k u \quad \text{(N-PVAR)}$$

$$\frac{\Downarrow^k P}{\Downarrow^k (\lambda x\text{:}A.\ P)} \quad \text{(N-ABS)} \qquad \frac{\Downarrow^k P}{\Downarrow^k (\mathbf{box}\ P)} \quad \text{(N-BOX)}$$

$$\frac{\Downarrow^k P \qquad \bigtriangledown^k P \qquad \Downarrow^k Q}{\Downarrow^k (P\ Q)} \quad \text{(N-APP)}$$

$$\frac{\Downarrow^{k-1} P}{\Downarrow^k (\mathbf{next}\ P)} \quad \text{(N-NEXT)} \qquad \frac{\Downarrow^{k-i} P \qquad \bigtriangledown^{k-i} P \qquad \Downarrow^k Q}{\Downarrow^k (\mathbf{let\ box}\ u =_i P\ \mathbf{in}\ Q)} \quad \text{(N-LETBOX)}$$

Figure 4. Neutral terms and normal forms

To prove this theorem, we need to show the property that any reduction at time k for M (such that $\Downarrow^k M$) does not yield a new redex at an earlier time, or formally:

If $\Delta; \Gamma \vdash^n M : A$ and $\Downarrow^k M$ and $M \xrightarrow{k} M'$, then $\Downarrow^k M'$.

This property, unfortunately, as it is cannot be proved by induction on $M \xrightarrow{k} M'$. A problematic case is when $M \equiv P\ Q \xrightarrow{k} P'\ Q \equiv M'$ is derived from $P \xrightarrow{k} P'$. It must not happen that P' is a λ-abstraction and $k > 0$ (otherwise $P'\ Q \xrightarrow{0} R$ for some R), but the induction hypothesis would not give such a guarantee. So we should take neutral terms into account and strengthen the statement to get Theorem 5 below, which we can now prove by straightforward induction on $M \xrightarrow{k} M'$. Theorem 4 follows from Theorem 5 as a corollary.

THEOREM 5 (Reduction Preserves Normality and Neutrality). *If $\Delta; \Gamma \vdash^n M : A$ and $\Downarrow^k M$ and $M \xrightarrow{k} M'$, then $\Downarrow^k M'$ holds and $\bigtriangledown^k M$ implies $\bigtriangledown^k M'$.*

Proof. By induction on the derivation of $M \xrightarrow{k} M'$, with a case analysis of the last rule used. Similarly to subject reduction, we need a lemma that substitution preserves normality and neutrality:

1. If $\Delta; \Gamma, x\text{:}^{m+i} B \vdash^m M : A$ and $\Downarrow^k M$ and $\Delta; \Gamma \vdash^{m+i} N : B$ and $\Downarrow^{k-i} N$ and $k \leq i$, then, $\Downarrow^k ([N/x]M)$ holds and $\bigtriangledown^k M$ implies $\bigtriangledown^k ([N/x]M)$.
2. If $\Delta, u\text{::}^{m+i} B; \Gamma \vdash^m M : A$ and $\Downarrow^k M$ and $\Delta; \cdot \vdash^{m+i} N : B$ and $\Downarrow^{k-i} N$ and $k \leq i$, then, $\Downarrow^k ([N/u]M)$ holds and $\bigtriangledown^k M$ implies $\bigtriangledown^k ([N/u]M)$.

Both (1) and (2) are proved by induction on the structure of M.

Now, we show a few representative cases:

Case $M \equiv \mathbf{let\ box}\ u =_i \mathbf{box}\ P\ \mathbf{in}\ Q$ reduces to $M' \equiv [P/u]Q$ by R-LETBOX**:** (Then, k must be equal to i.) By T-LETBOX and T-BOX, there exists type B s.t. $\Delta; \cdot \vdash^{n+i} P : B$ and $\Delta, u\text{::}^{n+i} B; \Gamma \vdash^n Q : A$. By $\Downarrow^k M$ and N-LETBOX, we have $\Downarrow^{k-i} P$, $\bigtriangledown^{k-i} P$ and $\Downarrow^k Q$. Applying the lemma above, we have $\Downarrow^k M'$. If $k \leq 0$, it is trivial that $\bigtriangledown^k M$ implies $\bigtriangledown^k M'$; otherwise, it vacuously holds, since M is not neutral, then.

Case $M \equiv P\ Q$ reduces to $M' \equiv P'\ Q$, with $P \xrightarrow{k} P'$ by a congruence rule**:** By T-APP, there exists some type B s.t., $\Delta; \Gamma \vdash^n P : B \to A$ and $\Delta; \Gamma \vdash^n Q : B$. By $\Downarrow^k M$ and N-APP, we have $\Downarrow^k P$, $\bigtriangledown^k P$ and $\Downarrow^k Q$. By the induction hypothesis, we have $\Downarrow^k P'$ and $\bigtriangledown^k P'$. Therefore by N-APP, we have $\Downarrow^k (P'\ Q)$, i.e., $\Downarrow^k M'$. Moreover, we have $\bigtriangledown^k M' (\equiv P'\ Q)$. □

4. Mini-ML$^{\bigcirc\Box}$

In this section, we extend $\lambda^{\bigcirc\Box}$ with some basic types, conditional expressions, local definitions, and recursion to obtain Mini-ML$^{\bigcirc\Box}$, in which examples shown in Section 2.4 are expressible, and give its call-by-value reduction semantics.

4.1 Syntax

We choose our small language to have: (1) Boolean type **bool** and its literal values **true**, **false**, as well as **if**–**then**–**else** construct; (2) integer type **int** and its literal values $\ldots, \overline{-1}, \overline{0}, \overline{1}, \overline{2}, \ldots$, as well as arithmetic operations of subtraction, multiplication and equality comparison; (3) **fix** operator for recursive function definitions; and (4) **let** construct for local definitions. The definition of types and additional terms is shown below.

$$\begin{array}{llll} \text{types} & A, B & ::= & \mathbf{bool} \mid \mathbf{int} \mid A \to B \mid \bigcirc A \mid \Box A \\ \text{terms} & L, M, N & ::= & \cdots \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N \\ & & \mid & \overline{n} \mid M - N \mid M * N \mid M = N \\ & & \mid & \mathbf{fix}\ x\text{:}A.\ M \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \end{array}$$

4.2 Typing Rules

The typing rules for the additional part of our language are quite standard:

$$\Delta; \Gamma \vdash^n \mathbf{true} : \mathbf{bool} \quad \text{(T-TRUE)}$$

$$\Delta; \Gamma \vdash^n \mathbf{false} : \mathbf{bool} \quad \text{(T-FALSE)}$$

$$\Delta;\Gamma \vdash^n \overline{m} : \mathbf{int} \qquad \text{(T-INT)}$$

$$\frac{\Delta;\Gamma \vdash^n L : \mathbf{bool} \quad \Delta;\Gamma \vdash^n M : A \quad \Delta;\Gamma \vdash^n N : A}{\Delta;\Gamma \vdash^n \mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N : A} \qquad \text{(T-IF)}$$

$$\frac{\Delta;\Gamma \vdash^n M : \mathbf{int} \quad \Delta;\Gamma \vdash^n N : \mathbf{int}}{\Delta;\Gamma \vdash^n M - N : \mathbf{int}} \qquad \text{(T-MINUS)}$$

$$\frac{\Delta;\Gamma \vdash^n M : \mathbf{int} \quad \Delta;\Gamma \vdash^n N : \mathbf{int}}{\Delta;\Gamma \vdash^n M * N : \mathbf{int}} \qquad \text{(T-MULT)}$$

$$\frac{\Delta;\Gamma \vdash^n M : \mathbf{int} \quad \Delta;\Gamma \vdash^n N : \mathbf{int}}{\Delta;\Gamma \vdash^n M = N : \mathbf{bool}} \qquad \text{(T-EQ)}$$

$$\frac{\Delta;\Gamma, x{:}^n A \vdash^n M : A}{\Delta;\Gamma \vdash^n \mathbf{fix}\, x{:}A.\, M : A} \qquad \text{(T-FIX)}$$

$$\frac{\Delta;\Gamma \vdash^n M : A \quad \Delta;\Gamma, x{:}^n A \vdash^n N : B}{\Delta;\Gamma \vdash^n \mathbf{let}\ x = M\ \mathbf{in}\ N : B} \qquad \text{(T-LET)}$$

4.3 Call-By-Value Reduction Semantics

Now, we define the call-by-value reduction semantics of Mini-ML$^{\bigcirc\Box}$ in a standard manner using evaluation contexts [7]. Here, formalized is program execution at stage 0, but as we will show this definition suffices: if a program is of type $\bigcirc A$, the resulting quoted code, after unquoting, can run as a new program at stage 0.

First we define the notion of *values*, taking the meaning of code expressions into account, and then proceed to the definition of the evaluation contexts and call-by-value reduction relation.

4.3.1 Values

The definition of values $V^{(0)}$ is given by the following grammar:

DEFINITION 2 (Values).

$$\begin{aligned} V^{(0)}, W^{(0)} ::= {} & \mathbf{true} \mid \mathbf{false} \mid \overline{n} \mid \lambda x{:}A.\, P \mid \mathbf{next}\, V^{(1)} \mid \mathbf{box}\, P \\ & \mid \mathbf{let\ box}\, u =_{n+1} V^{(n+1)}\ \mathbf{in}\ W^{(0)} \end{aligned}$$

$$\begin{aligned} V^{(k)}, W^{(k)} \quad & (k \geq 1) \\ ::= {} & \mathbf{true} \mid \mathbf{false} \mid \overline{n} \mid x \mid u \mid \lambda x{:}A.\, V^{(k)} \mid V^{(k)}\ W^{(k)} \\ & \mid \mathbf{if}\ U^{(k)}\ \mathbf{then}\ V^{(k)}\ \mathbf{else}\ W^{(k)} \\ & \mid V^{(k)} - W^{(k)} \mid V^{(k)} * W^{(k)} \mid V^{(k)} = W^{(k)} \\ & \mid \mathbf{fix}\, x{:}A.\, V^{(k)} \mid \mathbf{let}\ x = V^{(k)}\ \mathbf{in}\ W^{(k)} \\ & \mid \mathbf{next}\, V^{(k+1)} \mid \mathbf{prev}\, V^{(k-1)} \quad (k \geq 2) \mid \mathbf{box}\, V^{(k)} \\ & \mid \mathbf{let\ box}\, u =_n V^{(k+n)}\ \mathbf{in}\ W^{(k)} \end{aligned}$$

Values are constants, function values, or code values ($\mathbf{next}\, V^{(1)}$ and $\mathbf{box}\, P$). Note that $V^{(k)}$ ($k \geq 1$), which represents a quoted code fragment at stage k, is not subject to execution at stage 0, thus every term constructor is included. The value of the form $\mathbf{let\ box}\, u =_{n+1} V^{(n+1)}\ \mathbf{in}\ W^{(0)}$, which may appear unfamiliar, can be seen as a code value with an environment which binds u to an expression $V^{(n+1)}$, whose execution is delayed until stage $n+1$. Also note that $V^{(1)}$ does *not* include $\mathbf{prev}\, V^{(0)}$; otherwise $\mathbf{prev\ next}\, W^{(1)}$, which should be reduced at stage 0 (cf. R-PREV), would be a value.

4.3.2 Evaluation Contexts

An evaluation context is a pseudo term that has a *hole* $[\cdot]$, which indicates the place where the reduction should be happen. In Mini-ML$^{\bigcirc\Box}$, the hole can be at not only stage 0 as is expected but also stage 1, due to the possible redex of the form $\mathbf{prev\ next} \cdots$ inside **next**. So, we introduce two kinds of holes—$[\cdot]$, a hole at stage 0, which expects redices for R-BETA, R-LETBOX etc. in it, and $\{\cdot\}$, a hole at stage 1, which expects those for R-PREV etc. in it. Then, left-to-write, call-by-value evaluation contexts $E^{(n)}$ at stage n are defined as follows:

DEFINITION 3 (Evaluation Contexts). *Evaluation contexts $E^{(n)}$ are defined as follows, where* **op** *stands for $-$, $*$, or $=$:*

$$\begin{aligned} E^{(0)} ::= {} & [\cdot] \mid E^{(0)}\ P \mid V^{(0)}\ E^{(0)} \mid \mathbf{next}\, E^{(1)} \\ & \mid \mathbf{let\ box}\, u =_n E^{(n)}\ \mathbf{in}\ P \\ & \mid \mathbf{let\ box}\, u =_{n+1} V^{(n+1)}\ \mathbf{in}\ E^{(0)} \\ & \mid E^{(0)}\ \mathbf{op}\ P \mid V^{(0)}\ \mathbf{op}\ E^{(0)} \\ & \mid \mathbf{if}\ E^{(0)}\ \mathbf{then}\ P\ \mathbf{else}\ Q \mid \mathbf{let}\ x = E^{(0)}\ \mathbf{in}\ P \end{aligned}$$

$$\begin{aligned} E^{(k)} \quad & (k \geq 1) \\ ::= {} & \{\cdot\} \quad (k = 1) \mid \lambda x{:}A.\, E^{(k)} \mid E^{(k)}\ P \mid V^{(k)}\ E^{(k)} \\ & \mid \mathbf{next}\, E^{(k+1)} \mid \mathbf{prev}\, E^{(k-1)} \mid \mathbf{box}\, E^{(k)} \\ & \mid \mathbf{let\ box}\, u =_n E^{(k+n)}\ \mathbf{in}\ P \\ & \mid \mathbf{let\ box}\, u =_n V^{(k+n)}\ \mathbf{in}\ E^{(k)} \\ & \mid E^{(k)}\ \mathbf{op}\ P \mid V^{(k)}\ \mathbf{op}\ E^{(k)} \mid \mathbf{fix}\, x{:}A.\, E^{(k)} \\ & \mid \mathbf{if}\ E^{(k)}\ \mathbf{then}\ P\ \mathbf{else}\ Q \\ & \mid \mathbf{if}\ V^{(k)}\ \mathbf{then}\ E^{(k)}\ \mathbf{else}\ Q \\ & \mid \mathbf{if}\ V^{(k)}\ \mathbf{then}\ W^{(k)}\ \mathbf{else}\ E^{(k)} \\ & \mid \mathbf{let}\ E^{(k)} = \mathbf{in}\ P \mid \mathbf{let}\ V^{(k)} = \mathbf{in}\ E^{(k)} \end{aligned}$$

Notice the side condition $(k = 1)$ for $\{\cdot\}$.

4.3.3 Evaluation Rules

The evaluation rules are shown in Figure 5. $P \rightsquigarrow P'$ means "term P CBV-reduces to P'," while the auxiliary judgment $R \rhd^k R'$ means "redex R reduces to R' at relative stage k."

EXAMPLE 3. *We show* `power_cb` *in Section 2.4 again.*

$$\begin{aligned} \texttt{power\_cb} : {} & \Box\,\mathbf{int} \to \bigcirc\,\Box(\mathbf{int} \to \mathbf{int}) \\ \equiv {} & \lambda n'{:}\Box\mathbf{int}.\, \mathbf{let\ box}\, u = n'\ \mathbf{in}\ \mathbf{next\ box}\, \lambda x{:}\mathbf{int}.\, \mathbf{prev}(\\ & (\mathbf{fix}\, p{:}\mathbf{int}\to\bigcirc\mathbf{int}.\, \lambda m{:}\mathbf{int}. \\ & \quad \mathbf{if}\ m = 0\ \mathbf{then\ next}\, 1 \\ & \quad\quad \mathbf{else\ next}(x * \mathbf{prev}(p\ (m-1)))) \\ & u) \end{aligned}$$

The evaluation of `power_cb` $(\mathbf{box}\, 2)$ *proceeds in the following way, where p' is an abbreviation of the part $(\mathbf{fix}\, p.\ \ldots)$ in the definition of* `power_cb`*, and underlines indicate the place of holes in the evaluation contexts:*

$$\begin{aligned} & \underline{\texttt{power\_cb}\ (\mathbf{box}\, 2)} \\ \rightsquigarrow\ & \underline{\mathbf{let\ box}\, u = \mathbf{box}\, 2\ \mathbf{in\ next\ box}\, \lambda x.\ \mathbf{prev}(p'\ u)} \\ \rightsquigarrow\ & \mathbf{next\ box}\, \lambda x.\ \mathbf{prev}(\underline{p'\ 2}) \\ \rightsquigarrow\ & \mathbf{next\ box}\, \lambda x.\ \mathbf{prev}(\underline{(\lambda m.\, \mathbf{if}\ m = 0\ \mathbf{then\ next}\, 1 \\ & \qquad\qquad \mathbf{else\ next}(x * \mathbf{prev}(p'\ (m-1))))\ 2}) \\ \rightsquigarrow^*\ & \mathbf{next\ box}\, \lambda x.\ \mathbf{prev\ next}(x * \mathbf{prev}(\underline{p'\ 1})) \\ \rightsquigarrow^*\ & \mathbf{next\ box}\, \lambda x.\ \mathbf{prev\ next}(x * \mathbf{prev\ next}(x * \underline{\mathbf{prev\ next}\, 1})) \\ \rightsquigarrow\ & \mathbf{next\ box}\, \lambda x.\ \mathbf{prev\ next}(x * \underline{\mathbf{prev\ next}(x * 1)}) \\ \rightsquigarrow\ & \mathbf{next\ box}\, \lambda x.\ \mathbf{prev\ next}(\underline{x * (x * 1)}) \\ \rightsquigarrow\ & \mathbf{next\ box}\, \lambda x.\ x * (x * 1) \end{aligned}$$

4.4 Properties

The call-by-value reduction defined above is indeed reduction at time 0 in the following sense:

THEOREM 6. *If $\Delta;\Gamma \vdash^0 P : A$ and $P \rightsquigarrow P'$ then $P \xrightarrow{0} P'$.*

Axioms:

$$(\lambda x{:}A.\, P)\; V^{(0)} \rhd^0 [V^{(0)}/x]P \qquad \text{(E-BETA)}$$

$$\textsf{let box}\, u =_0 \textsf{box}\, P \,\textsf{in}\, Q \rhd^0 [P/u]Q \qquad \text{(E-LETBOX)}$$

$$(\textsf{let box}\, u =_{n+1} U^{(n+1)} \,\textsf{in}\, V^{(0)})\; W^{(0)} \rhd^0 \textsf{let box}\, u =_{n+1} U^{(n+1)} \,\textsf{in}\, (V^{(0)}\; W^{(0)}) \qquad \text{(E-APPC)}$$

$$\textsf{let box}\, v =_0 (\textsf{let box}\, u =_{n+1} U^{(n+1)} \,\textsf{in}\, V^{(0)}) \,\textsf{in}\, R \rhd^0 \textsf{let box}\, u =_{n+1} U^{(n+1)} \,\textsf{in let box}\, v =_0 V^{(0)} \,\textsf{in}\, R \qquad \text{(E-LETBOXC)}$$

$$\textsf{prev}(\textsf{next}\, V^{(1)}) \rhd^{-1} V^{(1)} \qquad \text{(E-PREV)}$$

$$\textsf{prev}(\textsf{let box}\, u =_{n+1} U^{(n+1)} \,\textsf{in}\, V^{(0)}) \rhd^{-1} \textsf{let box}\, u =_n U^{(n+1)} \,\textsf{in}\, (\textsf{prev}\, V^{(0)}) \qquad \text{(E-PREVC)}$$

$$\textsf{if true then}\, P \,\textsf{else}\, Q \rhd^0 P \qquad \text{(E-IF-TRUE)}$$

$$\textsf{if false then}\, P \,\textsf{else}\, Q \rhd^0 Q \qquad \text{(E-IF-FALSE)}$$

$$\overline{m} - \overline{n} \rhd^0 \overline{m-n} \qquad \text{(E-MINUS)}$$

$$\overline{m} * \overline{n} \rhd^0 \overline{m \times n} \qquad \text{(E-MULT)}$$

$$\overline{n} = \overline{n} \rhd^0 \textsf{true} \qquad \text{(E-COMP-EQ)}$$

$$\overline{m} = \overline{n} \rhd^0 \textsf{false} \quad (m \neq n) \qquad \text{(E-COMP-NEQ)}$$

$$\textsf{fix}\, x{:}A.\, P \rhd^0 [(\textsf{fix}\, x{:}A.\, P)/x]P \qquad \text{(E-FIX)}$$

$$\textsf{let}\; x = V^{(0)} \,\textsf{in}\, Q \rhd^0 [V^{(0)}/x]Q \qquad \text{(E-LET)}$$

Rules:

$$\frac{R \rhd^0 R'}{E^{(n)}[R] \rightsquigarrow E^{(n)}[R']} \qquad \text{(E-0)}$$

$$\frac{R \rhd^{-1} R'}{E^{(n)}\{R\} \rightsquigarrow E^{(n)}\{R'\}} \qquad \text{(E-1)}$$

Figure 5. Call-by-value reduction rules

Moreover, if the given program is well typed, reduction never get stuck and results in a unique value when it terminates.

THEOREM 7 (Progress). *If* $\cdot;\cdot \vdash^0 P : A$, *then either*

1. P *is a value* $V^{(0)}$, *or,*
2. *there exists unique* P' *such that* $P \rightsquigarrow P'$.

By combining the theorems above and Theorem 1, we can finally show the theorem of *binding-time correctness* [5] that executing a program of type $\bigcirc A$ yields another program that can be typed at stage 0 (if it terminates). Here, we abbreviate a sequence of **let box** bindings with a vector notation and, for example,

$$\textsf{let box}\, \vec{u} =_{\vec{n}+1} \vec{V}^{(\vec{n}+1)} \,\textsf{in}$$

means

$$\begin{array}{c}\textsf{let box}\, u_1 =_{n_1+1} V_1^{(\vec{n}_1+1)} \,\textsf{in}\\ \vdots\\ \textsf{let box}\, u_m =_{n_m+1} V_m^{(\vec{n}_m+1)} \,\textsf{in}\ .\end{array}$$

THEOREM 8. *If* $\cdot;\cdot \vdash^0 P : \bigcirc A$ *and* $P \rightsquigarrow^* V^{(0)}$, *then* $V^{(0)} \equiv \textsf{let box}\, \vec{u} =_{\vec{n}+1} \vec{V}^{(\vec{n}+1)} \,\textsf{in next}\, V^{(1)}$ *and* $\cdot;\cdot \vdash^0 \textsf{let box}\, \vec{u} =_{\vec{n}} \vec{V}^{(\vec{n}+1)} \,\textsf{in}\, V^{(1)} : A$.

Notice that indices attached to **let box** is decreased by 1 after removing **next**.

5. Related Work

5.1 Typed Calculi based on Modal Logic

As already mentioned, Davies [5] developed a typed calculus $\lambda^{\bigcirc}$, whose type system corresponds to linear-time temporal logic only with $\bigcirc$. The basic idea is that formula $\bigcirc A$ in the logic should correspond to the type of code which will be executed at the next stage. The property of time-ordered normalization is first introduced here, even though its formulation is somewhat informal and limited (as discussed in Section 3.4.) The notion of persistent code was, however, not considered there.

Davies and Pfenning [6] developed a typed calculus $\lambda^{\Box}$, by extending the Curry-Howard isomorphism to the modal logic S4, where formula $\Box A$ in the logic is interpreted as the type of code which has no free variable in it. As seen in Section 2.4, although $\lambda^{\Box}$ is equipped with a mechanism for run-time code generation, it cannot manipulate code containing free variables and so generated code is not as efficient as that of $\lambda^{\bigcirc}$.

Our calculus $\lambda^{\bigcirc\Box}$, which includes both of the calculi $\lambda^{\bigcirc}$ and $\lambda^{\Box}$ as its subsets can handle $\bigcirc$ and $\Box$ in a single system, by means of extending the Curry-Howard isomorphism to the linear-time temporal logic with modality $\Box$. However, supporting convertibility between types $\Box\bigcirc A$ and $\bigcirc\Box A$ and time-ordered normalization property required non-trivial extensions including commuting conversions.

Miyamoto and Igarashi's calculus $\lambda_s^{\Box}$ [16], equipped with a security type system for *information flow analysis* [29], is based on the modal logic of *local validity*—$\Box_\ell A$ is read "*A holds in any possible world reachable from world* ℓ." A computational interpretation of such a locally valid proposition is the type of the expression of type A which is accessible at the security level ℓ or any higher level. A type $\Box_\ell A$ in $\lambda_s^{\Box}$ and a type $\bigcirc\cdots\bigcirc\Box A$ in $\lambda^{\bigcirc\Box}$ are very similar to each other, in the sense that both express the type of the terms which are available at a particular or any higher level. Furthermore, *noninterference*, one of the most important correctness properties of $\lambda_s^{\Box}$, stating that "a program input at a higher security level does not affect the program output at a lower level", which can be said, in other words, low level computation can be performed without any high level computation. Relations between multi-level programs and information flow analysis have been studied elsewhere [1, 2], though they do not use modal logic very explicitly.

Murphy et al. [28, 27] developed typed lambda calculi as type-theoretic foundations for distributed programming; they are based on modal logic S5 (with $\Box A$ and $\Diamond A$), in which the underlying reachability relation is an equivalence relation. They extend the Curry-Howard isomorphism to this logic, by interpreting possible worlds as network nodes, and formulas $\Box A$ and $\Diamond A$ as the types of mobile code (of type A), which can be executed at any node, and addresses of remote values (of type A), respectively. Thus, the resulting systems handle both mobility of code and locality of resources in a single framework. Both their calculi and $\lambda^{\bigcirc\Box}$ interpret $\Box A$ as the type of code which is *portable to any reachable worlds* (network nodes or computation stages), but the two calculi differ from each other in reachability of such worlds. The former takes equivalence relations to describe a network in which all nodes are fully connected and can communicate with each other equally. On the other hand, the latter takes partial (linear) orders to describe staged program execution, in which computation goes on in one direction and never goes back.

5.2 Other Type Systems for Staged Programs

There have been much work on type systems for safe staged programs, based on $\lambda^{\bigcirc}$ and $\lambda^{\Box}$.

MetaML [25], developed by Taha and Sheard, is an extension of $\lambda^{\bigcirc}$, and has open code types $\langle A \rangle$, which corresponds to type $\bigcirc A$ of $\lambda^{\bigcirc}$, as well as a construct **run** for code execution. The resulting type system, however, is not strong enough to ensure that a code expression containing free variables is never **run**. Moggi et al. [18] added closed code types $[A]$ to MetaML, resulting in AIM (An Idealized MetaML), a simpler type system with the ability of safe code execution. Benaissa et al. [3] later developed λ^{BN} with further refinement of AIM. They interpret the type $[A]$ as a closed *value* type (not limited to code type) and unify the two kinds of code types; in this system, the closed code type which has been separately categorized is now handled as a special case of open code. Those type systems allow to build efficient code (as in $\lambda^{\bigcirc}$ or $\lambda^{\bigcirc\Box}$), which is persistent and immediately runnable by run-time code generation.

Some aspects of those type systems can be explained in terms of $\lambda^{\bigcirc\Box}$. For example, a type judgment $\Gamma \vdash e : A^n$ in AIM roughly corresponds to a $\lambda^{\bigcirc\Box}$ type judgment $\Gamma; \cdot \vdash^n e : A$ where the ordinary context is empty. Thus, every variable in AIM is a persistent variable in $\lambda^{\bigcirc\Box}$, naturally supporting cross-stage persistence. (Function abstractions in AIM can be represented by the combination of λ and **let box**.) So, AIM could be decomposed to $\lambda^{\bigcirc\Box}$ and some pragmatically motivated extensions, for example, to allow a construct (**run**) to remove the type constructor $\bigcirc$ without moving to the next computation stage. Although AIM and λ^{BN} seem to have stronger expressiveness for staged programs than $\lambda^{\bigcirc\Box}$, as far as we know, $\lambda^{\bigcirc\Box}$ seems to be the first one that combines $\lambda^{\bigcirc}$ and $\lambda^{\Box}$ by taking into account the Curry-Howard isomorphism, which is useful to give a foundational account for type systems for multi-level programs.

More recently, environment classifiers [24, 4] have been proposed as a typing mechanism for staged programs. Here, computation stages are generalized from (totally ordered) natural numbers, as in AIM and $\lambda^{\bigcirc\Box}$, to a sequence of environment classifiers, which gives tree-structured stages. Although both $\lambda^{\bigcirc}$ and $\lambda^{\Box}$ can be represented by using environment classifiers, it is not clear that $\lambda^{\bigcirc\Box}$ is also representable with environment classifiers—in particular, we suspect commutativity of the two modalities $\bigcirc$ and $\Box$ would be lost due to the lack of totality in the order of stages.

Nanevski [19] started with $\lambda^{\Box}$ and introduced a new notion called *names*, similar to the symbols in Lisp, to develop the calculus $\nu^{\Box}$. The calculus allows the **box** code expression of $\lambda^{\Box}$ to include free appearances of newly generated names (not variables), so that manipulation of code can be carried out symbolically. As a result, it is made possible to generate code that is as efficient as in $\lambda^{\bigcirc}$.

Kim, Yi, and Calcagno [13] developed another type system for staged programs based on (the implicit version [6] of) $\lambda^{\Box}$. Their language uses textual substitution, rather than capture-avoiding, for manipulation of code fragments and realizes capture-avoiding substitution by combining with a construct for name generation. The type system is in fact closer to those of context calculi [22, 21, 10].

6. Concluding Remarks

We have presented a typed calculus $\lambda^{\bigcirc\Box}$, whose type system is based on a proof system for (intuitionistic) linear-time temporal logic, as a foundation of programming languages for multi-level generating extensions with persistent code. The calculus enjoys not only basic properties such as subject reduction, confluence, and strong normalization but also time-ordered normalization. Commuting conversions play an important role for time-ordered normalization to hold.

Our calculus, as mentioned in the previous section, includes both $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ as its subsets and handle both kind of code, persistent one and ephemeral one; it in particular can express runtime code generation using persistent code, just as in $\lambda^{\Box}$, or `eval`-like construct in Lisp, and furthermore enables combination of the features of both kind of code, as shown in Section 2.4, although some limitation remains as yet.

From a logical point of view, our proof system, extracted from the type system of $\lambda^{\bigcirc\Box}$, is interesting. A proof system for (classical) temporal logic is often formalized in the Hilbert style with axioms and rules for each modalities. A common set of axioms and rules of linear-time temporal logic, due to Stirling [23], is as follows:

axioms:	L1	classical tautologies
	L2	$\Box(A \to B) \to (\Box A \to \Box B)$,
	L3	$\bigcirc \neg A \leftrightarrow \neg \bigcirc A$,
	L4	$\bigcirc(A \to B) \to (\bigcirc A \to \bigcirc B)$,
	L5	$\Box A \to A \wedge \bigcirc \Box A$,
	L6	$\Box(A \to \bigcirc A) \to (A \to \Box A)$,
rules:	MP	if $A \to B$ and A, then B,
	RG	if A, then $\Box A$.

Although we can prove L2, L4 and L5 and both rules MP and RG are expressed in terms of T-APP and T-BOX, our proof system lacks axioms L3, which is related to $\neg$ (the type constructor that does not exist in $\lambda^{\bigcirc\Box}$), and L6, which realizes the principle of mathematical induction on time.

Thus, it is interesting to add the power of the induction axiom to $\lambda^{\bigcirc\Box}$ since such an extension would enable us to write *non-uniform* persistent code, which results in different code depending on which stage it is used. By a straightforward extension of the Brouwer-Heyting-Kolmogorov interpretation [26], the proof of $\Box A$ can be thought as a function that takes a time (or stage) as an argument and returns the proof of A at that time. In $\lambda^{\bigcirc\Box}$, the proof term **box** M of $\Box A$ is always a constant function: no matter which stage it is used (by **let box**), the obtained proof of A is M (see R-LETBOX). In the presence of the induction axiom, however, the proof term of $\Box A$ as a function would return different proof terms depending on which stage it is used. In fact, we have been working on such an extension but have found that it would require significant changes to the calculus.

While our proof system is weaker than a familiar logic, it seems stronger than an axiomatic system obtained by removing the axiom L6, since, by counter-model construction, it can be shown that neither $\bigcirc\Box A \to \Box\bigcirc A$ nor $\Box\bigcirc A \to \bigcirc\Box A$ is provable

in such a proof system. We conjecture that our proof system is intermediate and equivalent to the following axiomatic system in the sense that we can derive $\cdot;\cdot \vdash^0 A$ in our system iff A is provable in the axiomatic system below:

axioms: $(A \to B \to C) \to (A \to B) \to (A \to C)$,
$A \to B \to A$,
$\Box(A \to B) \to (\Box A \to \Box B)$,
$\bigcirc(A \to B) \to (\bigcirc A \to \bigcirc B)$,
$(\bigcirc A \to \bigcirc B) \to \bigcirc(A \to B)$,
$\Box A \to A$,
$\Box A \to \Box\Box A$,
$\Box A \to \bigcirc A$,
$\Box\bigcirc A \to \bigcirc\Box A$,
$\bigcirc\Box A \to \Box\bigcirc A$

rules: if $A \to B$ and A, then B,
if A, then $\Box A$.

The first two axioms are usual axioms for implication. The fifth axiom, the *converse* of the K axiom of modality $\bigcirc$, enforces times to be linearly ordered. In the Stirling's formalization it is expressed in the right-to-left direction of axiom L3. To our knowledge, this kind of logic has not been considered previously. It is easy to prove those axioms and rules are admissible in $\lambda^{\bigcirc\Box}$ but it is left for future work to prove the converse.

Finally, it is also left for interesting future work to design and implement a full-fledged programming language based on $\lambda^{\bigcirc\Box}$. The present call-by-value reduction semantics, unfortunately, does not seem to suggest efficient implementation, especially because of the presence of commuting conversions (E-LETBOXC and E-PREVC). Designing a suitable abstract machine with environments would be a first step.

Acknowledgments

We thank Hidehiko Masuhara and anonymous referees for comments to improve presentation. This work was supported in part by Grant-in-Aid for Scientific Research on Priority Areas Research No. 18049044 from MEXT of Japan.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, TX, January 1999.

[2] Gilles Barthe and Bernard P. Serpette. Partial evaluation and non-interference for object calculi. In *Proceedings of 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *Lecture Notes on Computer Science*, pages 53–67, Tsukuba, Japan, 1999.

[3] Zine El-Abidine Benaissa, Eugenio Moggi, Walid Taha, and Tim Sheard. Logical modalities and multi-stage programming. In *Proceedings of Workshop on Intuitionistic Modal Logics and Applications (IMLA'99)*, Trento, Italy, July 1999. Available from `http://www.disi.unige.it/person/MoggiE/publications.html`.

[4] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-like inference for classifiers. In David Schmidt, editor, *Proceedings of European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes on Computer Science*, pages 79–93, Barcelona, Spain, March/April 2004. Springer Verlag.

[5] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of 11th Annual Symposium on Logic in Computer Science (LICS'96)*, pages 184–195, 1996.

[6] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[7] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[8] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[9] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *Proceedings of Programming Languages, Implementations, Logics and Programs (PLILP'95)*, LNCS 982, pages 259–278, 1995.

[10] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1-2):249–272, 2001.

[11] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structure in Computer Science*, 7:507–541, 1997. Special issue containing selected papers presented at the *1995 Workshop on Logic, Domains, and Programming Languages*. Darmstadt, Germany.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[13] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 257–268, Charleston, SC, January 2006.

[14] Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proceedings of International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, volume 1281 of *Lecture Notes on Computer Science*, pages 165–190, Sendai, Japan, September 1997. Springer Verlag.

[15] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1), 1996.

[16] Kenji Miyamoto and Atsushi Igarashi. A modal foundation for secure information flow. In *Proceedings of Workshop on Foundations of Computer Security (FCS'04)*, pages 187–203, 2004.

[17] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 1991.

[18] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *Proceedings of European Symposium on Programming (ESOP'99)*, volume 1576 of *LNCS*, pages 193–207, 1999.

[19] Aleksandar Nanevski. Meta-programming with names and necessity. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 206–217, 2002. See also `http://www-2.cs.cmu.edu/~aleks/papers/necessity/techrep2.ps`.

[20] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[21] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. In *Proceedings of Fifth International Symposium on Functional and Logic Programming (FLOPS 2001)*, volume 2024 of *Lecture Notes on Computer Science*, pages 359–374, Tokyo, Japan, 2001. Springer Verlag.

[22] Masahiko Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. *Journal of Functional and Logic Programming*, 2002(4):1–41, 2002.

[23] Colin Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science*, volume 2, pages 477–563. Oxford University Press, 1992.

[24] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 26–37, New Orleans, LA, January 2003.

[25] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, pages 203–217, Amsterdam, The Netherlands, June 1997. ACM Press.

[26] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics: An Introduction*, volume 1. North-Holland, 1988.

[27] Tom Murphy VII, Karl Crary, and Robert Harper. Distributed control flow with classical modal logic. In *Proceedings of Conference of the European Association for Computer Science Logic (CSL'05)*, volume 3634 of *Lecture Notes on Computer Science*, pages 51–69, Oxford, UK, August 2005. Springer Verlag.

[28] Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of 19th Annual Symposium on Logic in Computer Science (LICS'04)*, pages 286–295, 2004.

[29] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.