A Logical Foundation for Environment Classifiers

Takeshi Tsukada[1] and Atsushi Igarashi[2]

[1] Tohoku University
[2] Kyoto University

Abstract. Taha and Nielsen have developed a multi-stage calculus λ^{α} with a sound type system using the notion of environment classifiers. They are special identifiers, with which code fragments and variable declarations are annotated, and their scoping mechanism is used to ensure statically that certain code fragments are closed and safely runnable.
In this paper, we investigate the Curry-Howard isomorphism for environment classifiers by developing a typed λ-calculus $\lambda^{\triangleright}$. It corresponds to multi-modal logic that allows quantification by transition variables—a counterpart of classifiers—which range over (possibly empty) sequences of labeled transitions between possible worlds. This interpretation will reduce the "run" construct—which has a special typing rule in λ^{α}—and embedding of closed code into other code fragments of different stages—which would be only realized by the cross-stage persistence operator in λ^{α}—to merely a special case of classifier application. We prove that $\lambda^{\triangleright}$ enjoys basic properties including subject reduction, confluence, and strong normalization and that the execution of a well-typed $\lambda^{\triangleright}$ program is properly staged. Finally, we show that the proof system augmented with a classical axiom is sound and complete with respect to a Kripke semantics of the logic.

1 Introduction

A number of programming languages and systems that support manipulation of programs as data [1–5] have been developed in the last two decades. A popular language abstraction in these languages consists of the Lisp-like *quasiquotation* mechanism to create and compose code fragments and a function to run them like `eval` in Lisp. For those languages and systems, a number of type systems for so-called "multi-stage" calculi have been studied [5–11] to guarantee safety of generated programs *even before the generating program runs*.

Among them, some seminal work on the principled design of type systems for multi-stage calculi is due to Davies [7] and Davies and Pfenning [8]. They discovered the Curry-Howard isomorphism between modal/temporal logics and multi-stage calculi by identifying (1) modal operators in modal logic with type constructors for code fragments treated as data and, in the case of temporal logic, (2) the notion of time with computation stages. For example, the calculus $\lambda^{\bigcirc}$ [7], which can be thought as a reformulation of Glück and Jørgensen's calculus for multi-level generating extensions [6] by using explicit quasiquote and unquote in the language, corresponds to a fragment of linear-time temporal logic (LTL) with the temporal operator "next" (written $\bigcirc$) [12]. Here, linearly ordered time corresponds to the level of nesting of quasiquotations, and a modal formula $\bigcirc A$ to the type of *code* of type A. It, however, does not treat `eval`; in fact, the code type in $\lambda^{\bigcirc}$ represents open code, that is, code that may have free variables, so simply adding `eval` to the calculus does not work—code execution may fail by unbound variables. The calculus $\lambda^{\Box}$ [8], on the other hand, corresponds to (intuitionistic) modal logic S4 (only with the necessity operator $\Box$), in which a formula $\Box A$ is considered the type of *closed code* of type A. It supports safe `eval` since every code is closed, but inability to deal with open code hampers generation of efficient code. The following work by Taha and others [5, 13, 14, 9, 15] sought various forms of combinations of the two systems above to develop expressive type systems for multi-stage calculi.

Finally, Taha and Nielsen [9] developed a multi-stage calculus λ^{α}, which was later modified to make type inference possible [15] and implemented as a basis of MetaOCaml. The calculus λ^{α} has a strong type system while supporting open code, `eval` (called `run`), and the mechanism called cross-stage persistence (CSP), which allows a value to be embedded in a code fragment evaluated later. For the type system, they introduced the notion of environment classifiers, which are special identifiers with which code fragments and variable declarations are annotated. A key idea is to reduce the closedness checking of a code fragment (which is useful to guarantee the safety of `eval`) to the freshness checking of a classifier. Unfortunately, however, correspondence to a logic is not clear for λ^{α} any longer, resulting in somewhat ad-hoc typing rules and complicated operational semantics, which would be difficult to adapt to different settings.

In this paper, we investigate the Curry-Howard isomorphism for environment classifiers by developing a typed λ-calculus $\lambda^{\triangleright}$. The new calculus corresponds to a multi-modal logic that allows quantification by *transition variables*—the counterpart of environment classifiers. Multiple modalities correspond to indexing of code types by classifiers and quantifiers to types for classifier abstractions, used to ensure freshness of classifiers. One of our key ideas is to set, in the Kripke semantics, classifiers to range over possibly empty *sequences* of labels, attached to the transition function on possible worlds. A pleasant effect of this interpretation is that it will reduce the `run` construct—which has a peculiar typing rule in λ^{α}—and embedding of closed code into other code fragments of different stages—which would be only realized by the CSP operator in λ^{α}—to merely a special case of classifier application. Our technical contributions are as follows:

- Identification of a modal logic that corresponds to environment classifiers;
- Development of a new typed λ-calculus $\lambda^{\triangleright}$, naturally emerged from the correspondence, with its syntax, operational semantics, and type system;
- Proofs of basic properties as a multi-stage calculus; and
- Proofs of soundness and completeness of the proof system (augmented with a classical axiom) with respect to a Kripke semantics of the logic.

One missing feature in $\lambda^{\triangleright}$ is CSP for all types of values but we do not think it is a big problem. First, CSP for primitive or function values is easy to add as a primitive (if one gives up printing code representation of functional values as in MetaOCaml). Second, as mentioned above, embedding closed code into code fragments of later stages is supported by a different means. It does not seem very easy to add CSP for open code to $\lambda^{\triangleright}$, but we think it is rarely needed.

Organization of the Paper. In Section 2, we review λ^{α} and informally describe how the features of its type system correspond to those of a logic. In Section 3, we define the multi-stage calculus $\lambda^{\triangleright}$ and prove basic properties including subject reduction, strong normalization, confluence, and the property that big-step semantics implements staged execution. In Section 4, we formally define (a classical version of) the logic that corresponds to $\lambda^{\triangleright}$ and prove soundness and completeness of the proof system with respect to a Kripke semantics. Lastly, we discuss related work and conclude. We omit proofs of the properties from the paper; a full version of the paper with proofs is available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/classifiers.html`.

2 Interpreting Environment Classifiers in a Modal Logic

In this section, we informally describe how environment classifiers can be interpreted in a modal logic. We start with reviewing Davies' $\lambda^{\bigcirc}$ [7] to get an intuition of how notions in a modal logic correspond to those in a multi-stage calculus. Then, along with reviewing main ideas of environment classifiers, we describe our logic informally and how our calculus $\lambda^{\triangleright}$ is different from λ^{α} by Taha and Nielsen [9].

2.1 $\lambda^{\bigcirc}$: Multi-Stage Calculus Based on LTL

Davies has developed the typed multi-stage calculus $\lambda^{\bigcirc}$, which corresponds to a fragment of LTL by the Curry-Howard isomorphism. It can be considered the λ-calculus with a Lisp-like quasiquotation mechanism. We first review linear-time temporal logic and the correspondence between the logic and the calculus.

Linear-time temporal logic is a sort of temporal logic, in which the truth of propositions may depend on discrete and linearly ordered time, i.e., a given time has a unique time that follows it. Some of the standard temporal operators are $\bigcirc$ (to mean "next"), $\Box$ (to mean "always"), and U (to mean "until"). Its Kripke semantics can be given by taking the set of natural numbers as possible worlds; then, for example, the semantics of $\bigcirc$ is given by: $n \Vdash \bigcirc\tau$ if and only if $n+1 \Vdash \tau$, where $n \Vdash \tau$ is the satisfaction relation, which means "τ is true at world n."

In addition to the usual Curry-Howard correspondence between propositions and types and between proofs and terms, Davies has pointed out additional correspondences between time and computation stages (i.e., levels of nested quotations) and between the temporal operator $\bigcirc$ and the type constructor meaning "the type of code of". So, for example, $\bigcirc\tau_1 \to \bigcirc\tau_2$, which means "if τ_1 holds at next time, then τ_2 holds at next time," is considered the type of functions that take a piece of code of type τ_1 and return code of type τ_2. According to this intuition, he has developed $\lambda^{\bigcirc}$, corresponding to the fragment of LTL only with $\bigcirc$.

$\lambda^{\bigcirc}$ has two new term constructors **next** M and **prev** M, which correspond to the introduction and elimination rules of $\bigcirc$, respectively. The type judgment of $\lambda^{\bigcirc}$ is of the form $\Gamma \vdash^n M : \tau$, where Γ is a context, M is a term, τ is a type (a proposition of LTL, only with $\bigcirc$) and n is a natural number indicating a stage. A context, which corresponds to assumptions, is a mapping from variables to pairs of a type and a natural number, since the truth of a proposition depends on time. The key typing rules are those for **next** and **prev**:

$$\frac{\Gamma \vdash^{n+1} M : \tau}{\Gamma \vdash^{n} \mathbf{next}\ M : \bigcirc\tau} \qquad \frac{\Gamma \vdash^{n} M : \bigcirc\tau}{\Gamma \vdash^{n+1} \mathbf{prev}\ M : \tau}.$$

The former means that, if M is of type τ at level $n+1$, then, at level n, **next** M is code of type τ; the latter is its converse. Computationally, **next** and **prev** can be considered quasiquote and unquote, respectively. So, in addition to the standard β-reduction, $\lambda^{\bigcirc}$ has the reduction rule $\mathbf{prev}\ (\mathbf{next}\ M) \longrightarrow M$, which cancels **next** by **prev**.

The code types in $\lambda^{\bigcirc}$ are often called open code types, since the quoted code may contain free variables, so naively adding the construct to "run" quoted code does not work, since it may cause unbound variable errors.

2.2 Multi-Modal Logic for Environment Classifiers

Taha and Nielsen [9] have introduced environment classifiers to develop λ^{α}, which has quasiquotation, run, and CSP with a strong type system. We explain how λ^{α} can be derived from $\lambda^{\bigcirc}$.[3] Environment classifiers are a special kind of identifiers with which code types and quoting are annotated: for each classifier α, there are a type constructor $\langle\tau\rangle^{\alpha}$ for code and a term constructor $\langle M\rangle^{\alpha}$ to quote M. Then, a stage is naturally expressed by a sequence of classifiers, and a type judgment is of the form $\Gamma \vdash^{A} M : \tau$, where natural numbers in a $\lambda^{\bigcirc}$ type judgment are replaced with sequences A of classifiers. So, the typing rules of quoting and unquoting (written $\tilde{}M$) in λ^{α} are given as follows:

$$\frac{\Gamma \vdash^{A\alpha} M : \tau}{\Gamma \vdash^{A} \langle M\rangle^{\alpha} : \langle\tau\rangle^{\alpha}} \qquad \frac{\Gamma \vdash^{A} M : \langle\tau\rangle^{\alpha}}{\Gamma \vdash^{A\alpha} \tilde{}M : \tau}.$$

Obviously, this is a generalization of $\lambda^{\bigcirc}$: if only one classifier is allowed, then the calculus is essentially $\lambda^{\bigcirc}$.

The corresponding logic would also be a generalization of LTL, in which there are several "dimensions" of linearly ordered time. A Kripke frame for the logic is

[3] Unlike the original presentation, classifiers do not appear explicitly in contexts here. The typing rules shown are accordingly adapted.

given by a transition system [12] in which each transition relation is a map. More formally, a frame is a triple $(S, L, \{\xrightarrow{\alpha} \mid \alpha \in L\})$ where S is the (non-empty) set of states, L is the set of labels, and $\xrightarrow{\alpha} \in S \to S$ for each $\alpha \in L$. Then, the semantics of $\langle\tau\rangle^\alpha$ is given by: $s \Vdash \langle\tau\rangle^\alpha$ if and only if $s' \Vdash \tau$ for $s \xrightarrow{\alpha} s'$, where s and s' are states.

The calculus λ^α has also a scoping mechanism for classifiers and it plays a central role to guarantee safety of **run**. The term $(\alpha)M$, which binds α in M, declares that α is used locally in M and such a local classifier can be instantiated with another classifier by term $M[\beta]$. We show typing rules for them with one for **run** below:

$$\frac{\Gamma \vdash^A M : \tau \qquad \alpha \notin \mathrm{FV}(\Gamma, A)}{\Gamma \vdash^A (\alpha)M : (\alpha)\tau} \qquad \frac{\Gamma \vdash^A M : (\alpha)\tau}{\Gamma \vdash^A M[\beta] : \tau[\alpha := \beta]} \qquad \frac{\Gamma \vdash^A M : (\alpha)\langle\tau\rangle^\alpha}{\Gamma \vdash^A \mathbf{run}\ M : (\alpha)\tau}.$$

The rule for $(\alpha)M$ requires that α does not occur in the context—the term M has no free variable labeled α—and gives a type of the form $(\alpha)\tau$, which Taha and Nielsen called α-closed type, which characterizes a relaxed notion of closedness. The rule for **run** M says that an α-closed code fragment annotated with α can be run. Note that $\langle\cdot\rangle^\alpha$ (but not $(\alpha)\cdot$) is removed in the type of **run** M. Taha and Nielsen have shown that α-closedness is sufficient to guarantee safety of **run**.

When this system is to be interpreted as logic, it is fairly clear that $(\alpha)\tau$ is a kind of universal quantifier, as Taha and Nielsen has also pointed out [9]. Then, the question is "What does a classifier range over?", which has not really been answered so far. Another interesting question is "How can the typing rule for **run** be read logically?"

One plausible answer to the first question is that "classifiers range over the set of transition labels". This interpretation matches the rule for $M[\beta]$ and it seems that the typing rules without **run** (with a classical axiom) are sound and complete with the Kripke semantics that defines $s \Vdash (\alpha)\tau$ by $s \Vdash \tau[\alpha := \beta]$ for all $\beta \in L$. However, it is then difficult to explain the rule for **run**.

The key idea to solve this problem is to have classifiers range over the set of finite (and possibly empty) *sequences* of transition labels and to allow a classifier abstraction $(\alpha)M$ to be applied to also sequences of classifiers. Then, **run** will be unified to a special case of application of a classifier abstraction to the *empty* sequence. More concretely, we change the term $M[\beta]$ to $M[B]$, where B is a possibly empty sequence of classifiers (the left rule below). When B is empty and τ is $\langle\tau_0\rangle^\alpha$ (assuming τ_0 do not include α), the rule (as shown as the right rule below) can be thought as the typing rule of (another version of) **run**, since α-closed code of τ_0 becomes simply τ_0 (without $(\alpha)\cdot$ as in the original λ^α).

$$\frac{\Gamma \vdash^A M : (\alpha)\tau}{\Gamma \vdash^A M[B] : \tau[\alpha := B]} \qquad \frac{\Gamma \vdash^A M : (\alpha)\langle\tau_0\rangle^\alpha}{\Gamma \vdash^A M[\varepsilon] : \tau_0}$$

Another benefit of this change is that cross-stage persistence for closed code (or embedding of persistent code [10]) can be easily expressed. For example, if x is of the type $(\alpha)\langle\mathbf{int}\rangle^\alpha$, then it can be used as code computing an integer at *different* stages as in, say, $\langle \cdots (\,\tilde{}\, x[\alpha]) + 3 \cdots \langle \cdots 4 + (\,\tilde{}\tilde{}\, x[\alpha\beta]) \cdots \rangle^\beta \cdots \rangle^\alpha$. So, once a programmer obtains closed code, she can use it at any later stage.

Correspondingly, the semantics is now given by $v, \rho; s \Vdash \tau$ where v is a valuation for propositional variables and ρ is a mapping from classifiers to sequences of transition labels. Then, $v, \rho; s \Vdash \langle\tau\rangle^\alpha$ is defined by $v, \rho; s' \Vdash \tau$ where s' is reachable from s through the sequence $\rho(\alpha)$ of transitions and $v, \rho; s \Vdash (\alpha)\tau$ by: $v, \rho[A/\alpha]; s \Vdash \tau$ for any sequence A of labels ($\rho[A/\alpha]$ updates the value of α to be A). In Section 4, we give the formal definition of the Kripke semantics and show that the proof system, based in the ideas above, with double negation elimination is sound and complete to the semantics.

3 The Calculus $\lambda^{\triangleright}$

In this section, we define the calculus $\lambda^{\triangleright}$, based on the ideas described in the previous section: we first define its syntax, type system, and small-step full reduction semantics and states some basic properties; then we define big-step call-by-value semantics and shows that staged execution is possible with this semantics. Finally, we give an example of programming in $\lambda^{\triangleright}$. We intentionally make notations for type and term constructors different from λ^α because their precise meanings are different; it is also to avoid confusion when we compare the two calculi.

3.1 Syntax

Let Σ be a countably infinite set of *transition variables*, ranged over by α and β. A *transition*, denoted by A and B, is a finite sequence of transition variables; we write ε for the empty sequence and AB for the concatenation of the two transitions. We write Σ^* for the set of transitions. A transition is often called a stage. We write $\mathrm{FTV}(A)$ for the set of transition variables in A, defined by $\mathrm{FTV}(\alpha_1\alpha_2\ldots\alpha_n) = \{\alpha_i \mid 1 \le i \le n\}$.

Let PV be the set of base types (corresponding to propositional variables), ranged over by b. The set Φ of *types*, ranged over by τ and σ, is defined by the following grammar:

$$\textit{Types} \quad \tau ::= b \mid \tau \to \tau \mid \triangleright_\alpha \tau \mid \forall\alpha.\tau \ .$$

A type is a base type, a function type, a code type, which corresponds to $\langle\cdot\rangle^\alpha$ of λ^α, or an α-closed type, which corresponds to $(\alpha)\tau$. The transition variable α of $\forall\alpha.\tau$ is bound in τ. In what follows, we assume tacit renaming of bound variables in types. The type constructor $\triangleright_\alpha$ connects tighter than $\to$ and $\to$ tighter than $\forall$: for example, $\triangleright_\alpha\tau \to \sigma$ means $(\triangleright_\alpha\tau) \to \sigma$ and $\forall\alpha.\tau \to \sigma$ means $\forall\alpha.(\tau \to \sigma)$. We write $\mathrm{FTV}(\tau)$ for the set of free transition variables, which is defined in a straightforward manner.

Let Υ be a countably infinite set of *variables*, ranged over by x and y. The set of *terms*, ranged over by M and N, is defined by the following grammar:

$$\textit{Terms} \quad M ::= x \mid M\, M \mid \lambda x : \tau.M \mid \blacktriangleright_\alpha M \mid \blacktriangleleft_\alpha M \mid \Lambda\alpha.M \mid M\ A \ .$$

In addition to the standard λ-terms, there are four more terms, which correspond to $\langle M\rangle^\alpha$, $\tilde{}M$, $(\alpha)M$, and $M[\beta]$ of λ^α (respectively, in the order presented). Note

$$\frac{}{\Gamma, x : \tau@A \vdash^A x : \tau} \ (\textsc{Var})$$

$$\frac{\Gamma, x : \tau@A \vdash^A M : \sigma}{\Gamma \vdash^A \lambda x : \tau.M : \tau \to \sigma} \ (\textsc{Abs}) \qquad \frac{\Gamma \vdash^A M : \tau \to \sigma \qquad \Gamma \vdash^A N : \tau}{\Gamma \vdash^A M\,N : \sigma} \ (\textsc{App})$$

$$\frac{\Gamma \vdash^{A\alpha} M : \tau}{\Gamma \vdash^A \blacktriangleright_\alpha M : \triangleright_\alpha \tau} \ (\blacktriangleright) \qquad \frac{\Gamma \vdash^A M : \triangleright_\alpha \tau}{\Gamma \vdash^{A\alpha} \blacktriangleleft_\alpha M : \tau} \ (\blacktriangleleft)$$

$$\frac{\Gamma \vdash^A M : \tau \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(A)}{\Gamma \vdash^A \Lambda\alpha.M : \forall\alpha.\tau} \ (\textsc{Gen}) \qquad \frac{\Gamma \vdash^A M : \forall\alpha.\tau}{\Gamma \vdash^A M\,B : \tau[\alpha := B]} \ (\textsc{Ins})$$

Fig. 1. Typing rules.

that, unlike ˜M in λ^α, the term $\blacktriangleleft_\alpha M$ for unquote is also annotated. The variable x in $\lambda x : \tau.M$ and the transition variable α in $\Lambda\alpha.M$ are bound in M. Bound variables are tacitly renamed to avoid variable capture in substitution.

3.2 Type System

As mentioned above, a type judgment and variable declarations in a context are annotated with stages. A *context* Γ is a finite set $\{x_1 : \tau_1@A_1, \ldots, x_n : \tau_n@A_n\}$, where x_i are distinct variables. We often omit braces $\{\}$. We write $\mathrm{FTV}(\Gamma)$ for the set of free transition variables in Γ, defined by: $\mathrm{FTV}(\{x_i : \tau_i@A_i \mid 1 \le i \le n\}) = \bigcup_{i=1}^{n}(\mathrm{FTV}(\tau_i) \cup \mathrm{FTV}(A_i))$.

A type judgment is of the form $\Gamma \vdash^A M : \tau$, read "term M is given type τ under context Γ at stage A." Figure 1 presents the typing rules to derive type judgments. The notation $\tau[\alpha := B]$, used in the rule (Ins), is capture-avoiding substitution of transition B for α in τ. When α in $\triangleright_\alpha$ is replaced by a transition, we identify $\triangleright_\varepsilon \tau$ with τ and $\triangleright_{AB}\tau$ with $\triangleright_A \triangleright_B \tau$. For example, $(\triangleright_\alpha \forall\alpha. \triangleright_\alpha b)[\alpha := \varepsilon] = \forall\alpha. \triangleright_\alpha b$ and $(\forall\alpha. \triangleright_\beta b)[\beta := \alpha\alpha] = \forall\alpha'. \triangleright_\alpha \triangleright_\alpha b$.

The first three rules are almost standard except for the stage annotations, which must be equal as in most multi-stage calculi. The rule (Var) means that variables can appear only at the stage which variables are declared. The next two rules ($\blacktriangleright$) and ($\blacktriangleleft$) are for quoting and unquoting and already explained in the previous section. The last two rules (Gen) and (Ins) are for generalization and instantiation of a transition variable, respectively. They resemble the introduction and elimination rules of $\forall x.A(x)$ in first-order predicate logic: the side condition of the (Gen) rule ensures that the choice of α is independent of the context. Computationally, this side condition expresses α-closedness of M, that means M has no free variable which has annotation α in its type or its stage. This is a weaker form of closedness, which means M has no free variable at all.

3.3 Reduction

We will introduce full reduction $M \longrightarrow N$, read "M reduces to N in one step," and prove basic properties including subject reduction, confluence and strong normalization.

Before giving the definition of reduction, we define substitution. Since the calculus has binders for term variables and transition variables, we need two kinds of substitutions for both kinds of variables. Substitution $M[x := N]$ for a term variable is the standard capture-avoiding one, and its definition is omitted here. Substitution $M[\alpha := A]$ of A for α is defined similarly to $\tau[\alpha := A]$. For example, $(\lambda x : \tau.M)[\alpha := A] = \lambda x : (\tau[\alpha := A]).(M[\alpha := A])$, $(M\ B)[\alpha := A] = (M[\alpha := A])\,(B[\alpha := A])$ and $(\blacktriangleright_\beta M)[\alpha := A] = \blacktriangleright_{\beta[\alpha:=A]}(M[\alpha := A])$, where we define $\blacktriangleright_{\alpha_1\ldots\alpha_n} M = \blacktriangleright_{\alpha_1} \cdots \blacktriangleright_{\alpha_n} M$ and $\blacktriangleleft_{\alpha_1\ldots\alpha_n} M = \blacktriangleleft_{\alpha_n} \cdots \blacktriangleleft_{\alpha_1} M$. In particular, $(\blacktriangleright_\alpha M)[\alpha := \varepsilon] = (\blacktriangleleft_\alpha M)[\alpha := \varepsilon] = M[\alpha := \varepsilon]$. Note that, when a transition variable in $\blacktriangleleft$ is replaced, the order of transition variables is reversed, because this is the inverse operation of $\blacktriangleright$. This is similar to the inversion operation in group theory: $(a_1 a_2 \ldots a_n)^{-1} = a_n^{-1} a_{n-1}^{-1} \ldots a_1^{-1}$.

The reduction relation $M \longrightarrow N$ is the least relation closed under the following three computation rules

$$(\lambda x.M)N \longrightarrow M[x := N] \qquad \blacktriangleleft_\alpha(\blacktriangleright_\alpha M) \longrightarrow M \qquad (\Lambda\alpha.M)A \longrightarrow M[\alpha := A]$$

and congruence rules, which are omitted here. In addition to the standard β-reduction, there are two rules: the second one, which is already explained previously, cancels quote by unquote and the last one, instantiation of a transition variable, is similar to polymorphic function application in System F. Note that the reduction is full—reduction occurs under any context—and does not take staging into account. We can define the reduction relation as a triple $M \xrightarrow{T} N$, with T standing for the stage of reduciton, as done in $\lambda^{\bigcirc}$ [7] and $\lambda^{\bigcirc\Box}$ [10].

The reduction enjoys three basic properties, subject reduction, strong normalization and confluence.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash^A M : \tau$ and $M \longrightarrow M'$, then $\Gamma \vdash^A M' : \tau$.*

Theorem 2 (Strong Normalization). *Let M be a typable term. There is no infinite reduction sequence $M \longrightarrow N_1 \longrightarrow N_2 \longrightarrow \cdots$.*

Theorem 3 (Confluence). *If $M \longrightarrow^* N_1$ and $M \longrightarrow^* N_2$, then there exists N such that $N_1 \longrightarrow^* N$ and $N_2 \longrightarrow^* N$.*

3.4 Big-Step Semantics

Now, we give a big-step semantics and prove that the execution of a well-typed program can be properly divided into stages. The judgment has the form $\vdash^A M \Downarrow R$, read "evaluating term M of stage A yields result R," where R is either **err**, which stands for a run-time error, or a value v, defined below. Values are given via a family of sets V^A indexed by transitions, that is, stages. The family V^A is defined by the following grammar:

$$\begin{array}{ll} V^\varepsilon & ::= \lambda x : \tau.M \mid \blacktriangleright_\alpha V^\alpha \mid \Lambda\alpha.V^\varepsilon \\ V^A\ (A \neq \varepsilon) & ::= x \mid \lambda x : \tau.V^A \mid V^A V^A \mid \blacktriangleright_\alpha V^{A\alpha} \mid \Lambda\alpha.V^A \mid V^A B \\ & \quad \mid \blacktriangleleft_\alpha V^{A'} \quad (\text{if } A'\alpha = A \text{ and } A' \neq \varepsilon) \end{array}$$

$$\frac{}{\vdash^{\varepsilon} \lambda x:\tau.M \Downarrow \lambda x:\tau.M} \qquad \frac{\vdash^{\varepsilon} M \Downarrow \lambda x:\tau.M' \qquad \vdash^{\varepsilon} N \Downarrow v \qquad \vdash^{\varepsilon} M'[x:=v] \Downarrow v'}{\vdash^{\varepsilon} M\,N \Downarrow v'}$$

$$\frac{\vdash^{\varepsilon} M \Downarrow \blacktriangleright_{\alpha} M'}{\vdash^{\alpha} \blacktriangleleft_{\alpha} M \Downarrow M'} \qquad \frac{\vdash^{\varepsilon} M \Downarrow \Lambda\alpha.v \qquad \vdash^{\varepsilon} v[\alpha := B] \Downarrow v'}{\vdash^{\varepsilon} M\,B \Downarrow v'} \qquad \frac{\vdash^{B} M \Downarrow M'}{\vdash^{B} \Lambda\alpha.M \Downarrow \Lambda\alpha.M'}$$

$$\frac{\vdash^{B\alpha} M \Downarrow M'}{\vdash^{B} \blacktriangleright_{\alpha} M \Downarrow \blacktriangleright_{\alpha} M'} \qquad \frac{}{\vdash^{A} x \Downarrow x} \qquad \frac{\vdash^{A} M \Downarrow M'}{\vdash^{A} \lambda x:\tau.M \Downarrow \lambda x:\tau.M'}$$

$$\frac{\vdash^{A} M \Downarrow M' \qquad \vdash^{A} N \Downarrow N'}{\vdash^{A} M\,N \Downarrow M'\,N'} \qquad \frac{\vdash^{A} M \Downarrow M'}{\vdash^{A\alpha} \blacktriangleleft_{\alpha} M \Downarrow \blacktriangleleft_{\alpha} M'} \qquad \frac{\vdash^{A} M \Downarrow M'}{\vdash^{A} M\,B \Downarrow M'\,B}$$

Fig. 2. Big-Step Semantics. Here, A stands for a non-empty sequence and B for a possibly empty sequence of transition variables.

The set V of values is defined as $\bigcup_{A \in \Sigma^*} V^A$.

Figure 2 shows the evaluation rules. The evaluation is left-to-right, call-by-value. The first six rules (where $B = \varepsilon$) are for ordinary evaluation. The first two rules are standard. The third rule means that quote is canceled by unquote; since the resulting term M' belongs to the stage α (inside quotation), α is attached to the conclusion. The fourth rule about instantiation of a transition abstraction is straightforward. As seen in the fifth rule for $\Lambda\alpha.M$, Λ does *not* delay the evaluation of the body. The rules for stages later than ε are all similar: since the term to be evaluated is inside quotation, the term constructor is left as it is and only subterms of stage ε are evaluated. For brevity, we do not present the error-generating rules and the error-propagating rules, which are straightforward.

We show properties of the big-step semantics. The following theorem says that, unless the result is **err**, the result must be a value even though the rules do not say it is the case, and that the successful evaluation is included in multi-step reduction ($\longrightarrow^*$ stands for the reflexive transitive closure of $\longrightarrow$).

Theorem 4. *Suppose $\vdash^A M \Downarrow R$. Then, either $R = \textbf{err}$ or $M \longrightarrow^* R \in V^A$.*

The last property is type soundness and its corollary that if a well-typed program of a code type yields a result, then the result is a quoted term, whose body is also typable at stage ε. In the statements, we say Γ is ε-free if it satisfies $A \neq \varepsilon$ for any $x : \tau@A \in \Gamma$ and define a context Γ^{-A} by: $\Gamma^{-A} = \{x : \tau@B \mid x : \tau@AB \in \Gamma\}$.

Theorem 5 (Type Soundness). *If Γ is ε-free and $\Gamma \vdash^{\varepsilon} M : \tau$ and $\vdash^{\varepsilon} M \Downarrow R$, then $R = v$ and $v \in V^{\varepsilon}$ for some v and $\Gamma \vdash^{\varepsilon} v : \tau$. Moreover, if $\tau = \triangleright_{\alpha}\tau_0$, then $v = \blacktriangleright_{\alpha} N$ and $\Gamma^{-\alpha} \vdash^{\varepsilon} N : \tau_0$.*

3.5 Programming in $\lambda^{\triangleright}$

We give an example of programming in $\lambda^{\triangleright}$. The example is the power function, which is a classical example in multi-stage calculi and partial evaluation. We augment $\lambda^{\triangleright}$ with integers, booleans, arithmetic and comparison operators, **if-then-else**, a fixed point operator **fix**, and **let**, all of which would be easy to add.

For readability, we often omit type annotations and put terms under quotation in shaded boxes.

We start with the ordinary power function without staging.

$$\begin{aligned}&\mathbf{let}\,\texttt{power}_0 :\ \mathbf{int} \to \mathbf{int} \to \mathbf{int}\\ &\quad = \mathbf{fix}\, f.\ \lambda n.\ \lambda x.\,\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x * (f\ (n-1)\ x)\end{aligned}$$

Our purpose is to get a code generator $\texttt{power}_\forall$ that takes the exponent n and returns (closed, hence runnable) code of $\lambda x.x * x * \ldots x * 1$, which computes x^n without recursion. Here, we follow the construction of code generator in the previous work [14, 13].

First, we construct a code manipulator $\texttt{power}_1 : \mathbf{int} \to \triangleright_\alpha \mathbf{int} \to \triangleright_\alpha \mathbf{int}$, which takes an integer n and a piece of integer code and then outputs a piece of code which connects the input code by "$*$" n times. It can be obtained by changing type annotation and introducing quasiquotation.

$$\begin{aligned}&\mathbf{let}\,\texttt{power}_1 :\ \mathbf{int} \to \triangleright_\alpha\mathbf{int} \to \triangleright_\alpha\mathbf{int}\\ &\quad = \mathbf{fix}\, f.\ \lambda n.\ \lambda x : \triangleright_\alpha\ \mathbf{int}.\\ &\qquad\quad \mathbf{if}\ n = 0\ \mathbf{then}\ (\blacktriangleright_\alpha 1)\ \mathbf{else}\ \blacktriangleright_\alpha ((\blacktriangleleft_\alpha x) * (\blacktriangleleft_\alpha f\ (n-1)\ x))\end{aligned}$$

Then, from $\texttt{power}_1$, we can construct a code generator $\texttt{power}_\alpha$ of type $\mathbf{int} \to \triangleright_\alpha(\mathbf{int} \to \mathbf{int})$, which means it takes an integer and returns code of a function.

$$\begin{aligned}&\mathbf{let}\,\texttt{power}_\alpha :\ \mathbf{int} \to \triangleright_\alpha(\mathbf{int} \to \mathbf{int})\\ &\quad = \lambda n.\ \blacktriangleright_\alpha \lambda x{:}\mathbf{int}.\ \blacktriangleleft_\alpha (\texttt{power}_1\ n\ (\blacktriangleright_\alpha x))\end{aligned}$$

It indeed behaves as a code generator: for example, $\texttt{power}_\alpha\ 3$ would evaluate to $\blacktriangleright_\alpha \lambda x : \mathbf{int}\,.x * (x * (x * 1))$.

This construction is independent of the choice of the stage α. So, by abstracting α at appropriate places in $\texttt{power}_1$ and $\texttt{power}_\alpha$, we can obtain the desired code generator, whose return type is a closed code type $\forall\alpha.\,\triangleright_\alpha\,(\mathbf{int} \to \mathbf{int})$.

$$\begin{aligned}&\mathbf{let}\,\texttt{power}_2 : \forall\alpha.\,\mathbf{int} \to \triangleright_\alpha\mathbf{int} \to \triangleright_\alpha\mathbf{int}\\ &\quad = \Lambda\alpha.\,\mathbf{fix}\, f.\ \lambda n.\ \lambda x : \triangleright_\alpha\ \mathbf{int}.\\ &\qquad\quad \mathbf{if}\ n = 0\ \mathbf{then}\ (\blacktriangleright_\alpha 1)\ \mathbf{else}\ \blacktriangleright_\alpha ((\blacktriangleleft_\alpha x) * (\blacktriangleleft_\alpha f\ (n-1)\ x))\\ &\mathbf{let}\,\texttt{power}_\forall : \mathbf{int} \to \forall\alpha.\ \triangleright_\alpha\ (\mathbf{int} \to \mathbf{int})\\ &\quad = \lambda n.\ \Lambda\alpha.\,\blacktriangleright_\alpha \lambda x{:}\mathbf{int}.\ \blacktriangleleft_\alpha (\texttt{power}_2\ \alpha\, n\ (\blacktriangleright_\alpha x))\end{aligned}$$

The output from $\texttt{power}_\forall$ is usable in any stage. For example, if we want code of a cube function at the stage A, we write $\texttt{power}_\forall\ 3\ A$. In particular, when A is the empty sequence ε, $\texttt{power}_\forall\ 3\, \varepsilon : \mathbf{int} \to \mathbf{int}$ evaluates to a function closure which computes $x * x * x * 1$ from the input x.

4 Kripke Semantics for $\lambda^{\triangleright}$ and Logical Completeness

In this section, we formally define a Kripke semantics of the logic corresponding to $\lambda^{\triangleright}$ and prove completeness of the proof system. Actually, what we examine here is a classical version of the logic, which has bottom and a proof rule for double negation elimination, although $\lambda^{\triangleright}$ itself can be considered intuitionistic. It is left for future work to study the semantics of the intutionistic version, of which recent work on Kripke semantics for intuitionistic LTL [16] can be a basis.

First, we (re)define the set of propositions and the natural deduction proof system. Then, we proceed to the formal definition of the Kripke semantics and state soundness and completeness of the proof system.

4.1 Natural Deduction

The set $\Phi_{\perp}$, ranged over by ϕ and ψ, of propositions are given by the grammar for Φ extended with a new constant $\perp$.

The natural deduction system can be obtained by forgetting variables and terms in the typing rules. We add the following new rule, which is the ordinary double negation elimination rule, adapted for this setting:

$$\frac{\Gamma, (\phi \to \perp)@A \vdash^{B} \perp}{\Gamma \vdash^{A} \phi}\ (\perp\text{-E}) \quad .$$

4.2 Kripke Semantics and Completeness

As mentioned in Section 2, the Kripke semantics for this logic is based on a functional transition system $\mathcal{T} = (S, L, \{\xrightarrow{a} \mid a \in L\})$ where S is the (non-empty) countable set of states, L is the countable set of labels, and $\xrightarrow{a} \in S \to S$ for each label $a \in L$. We write $s \xrightarrow{a_1 \cdots a_n} s'$ if there exist $s_1, \ldots, s_{n-1}$ such that $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s'$.

To interpret a proposition, we need two valuations, one for propositional variables and the other for transition variables. The former is a total function $v \in S \times \mathrm{PV} \to \{0, 1\}$; the latter is a total function $\rho \in \Sigma \to L^*$, where L^* is the set of all finite sequences of labels. Then, we define the satisfaction relation $\mathcal{T}, v, \rho; s \Vdash \phi$, where $s \in S$ is a state, as follows:

$$\begin{array}{lll}
\mathcal{T}, v, \rho; s \Vdash p & \text{iff} & v(s, p) = 1 \\
\mathcal{T}, v, \rho; s \Vdash \perp & & \text{never occurs} \\
\mathcal{T}, v, \rho; s \Vdash \phi \to \psi & \text{iff} & \mathcal{T}, v, \rho; s \nVdash \phi \ \text{ or } \ \mathcal{T}, v, \rho; s \Vdash \psi \\
\mathcal{T}, v, \rho; s \Vdash \triangleright_{\alpha}\phi & \text{iff} & \mathcal{T}, v, \rho; s' \Vdash \phi \text{ where } s \xrightarrow{\rho(\alpha)} s' \\
\mathcal{T}, v, \rho; s \Vdash \forall\alpha.\phi & \text{iff} & \text{for all } A \in L^*, \ \ \mathcal{T}, v, \rho[A/\alpha]; s \Vdash \phi
\end{array}$$

Here, $\rho[A/\alpha]$ is defined by: $\rho[A/\alpha](\alpha) = A$ and $\rho[A/\alpha](\beta) = \rho(\beta)$ (for $\beta \neq \alpha$). The satisfaction relation is extended pointwise to contexts Γ (possibly infinite sets of pairs of a proposition and a transition) by:

$$\mathcal{T}, v, \rho; s \Vdash \Gamma \ \text{ iff } \ \mathcal{T}, v, \rho; s \Vdash \triangleright_{A}\phi \text{ for all } \phi@A \in \Gamma \ .$$

The local consequence relation $\Gamma \Vdash \phi$ is defined by:

$$\Gamma \Vdash \phi \text{ iff } \mathcal{T}, v, \rho; s \Vdash \Gamma \text{ implies } \mathcal{T}, v, \rho; s \Vdash \phi \text{ for any } \mathcal{T}, v, \rho, s \ .$$

Then, the natural deduction proof system is sound and complete with respect to the local consequence relation. The proof is similar to the one for first-order predicate logic: we use the standard techniques of Skolemization and Herbrand structure.

Theorem 6. *$\Gamma \vdash^{\varepsilon} \phi$ if and only if $\Gamma \Vdash \phi$.*

5 Related Work

Multi-Stage Calculi Based on Modal Logics and Their Extensions. Our work can be considered a generalization of the previous work on the Curry-Howard isomorphism between multi-stage calculi and modal logics [7, 8, 10]. Here, we briefly discuss how the earlier systems $\lambda^{\bigcirc}$ and $\lambda^{\Box}$ can be embedded to $\lambda^{\triangleright}$.

First, as already mentioned in Section 2, $\lambda^{\bigcirc}$ is obtained by using only one transition variable; so, $\bigcirc$ translates to $\triangleright_\alpha$ with a fixed transition variable α; **next** and **prev** to $\blacktriangleright_\alpha$ and $\blacktriangleleft_\alpha$, respectively.

Second, the calculus $\lambda^{\Box}$ [8], which corresponds to intuitionistic modal logic S4 (with $\Box$). The type $\Box\tau$ represents closed code values, which thus can be run or embedded in code of any later stages, as is possible in $\lambda^{\triangleright}$. There are **box** and $\mathbf{unbox}_n$ for quoting and unquoting, respectively (see Pfenning and Davies [8] for details).[4] The $\lambda^{\Box}$-type $\Box\tau$ corresponds to $\forall\alpha.\triangleright_\alpha \tau$, where τ does not include α; so, it reflects the fact that the code type in $\lambda^{\Box}$ is (completely) closed. Unlike the embedding from $\lambda^{\Box}$ to λ^{α}, given in [9], there is no use of CSP.

The restriction of $\lambda^{\Box}$ that all code be closed precludes the definition of a code generator like $\texttt{power}_{\forall}$, which generates both efficient and runnable code. Nanevski and Pfenning [17] have extended $\lambda^{\Box}$ with the notion of names, similar to the symbols in Lisp, and remedied the defect of $\lambda^{\Box}$ by allowing newly generated names (not variables) to appear in closed code.

Taha and Sheard [5] added **run** and CSP to $\lambda^{\bigcirc}$ and developed MetaML, but its type system was not strong enough—**run** may fail at run-time. Then, Moggi, Taha, Benaissa, and Sheard [13] developed the calculus AIM ("An Idealized MetaML"), in which there are types for both open and closed code; it was simplified to λ^{BN}, which replaced closed code types with closedness types for closed (but not necessarily code) terms. Both calculi are based on categorical models and have sound type systems. The notion of α-closedness in λ^{α} can be considered a generalization of λ^{BN}'s closed types. In fact, the typing rule for **run** in λ^{BN} is similar to the one in λ^{α}. Although some of these calculi have sound type systems, it is hard to regard them as logic, mainly due to the presence of CSP, which delays the stage of the type judgment to any later stage, and the typing rule for **run** (as discussed in Section 2).

[4] Precisely speaking, this calculus is what they call the "Kripke-style" calculus.

One nice property of λ^α is that a program can be executed without exploiting information on classifiers; in other words, classifiers can be erased after typechecking. Although our calculus $\lambda^{\triangleright}$ does not have this "erasure property," due to the presence of abstraction/instantiation of transition variables, by restricting $\forall$-types to be of the form $\forall\alpha.\triangleright_\alpha\tau$ where $\alpha \notin \mathrm{FTV}(\tau)$, information on transition variables can be mostly erased. Under this restriction, the only information to be left after erasure is the length n of A in $M\,A$, which only duplicates $\blacktriangleright$ at the head of the value of M n times. This restriction, which resembles one in λ_i [15], still allows embedding of $\lambda^{\bigcirc}$ and $\lambda^{\Box}$ and $\mathtt{power}_\forall$ (by inlining $\mathtt{power}_2$ into the body of it).

Comparing λ^α and $\lambda^{\triangleright}$, we point out two differences between them. First, λ^α has CSP for all terms but $\lambda^{\triangleright}$ cannot express CSP for open code. While we can deal with CSP for closed code as syntactic sugar, CSP for open code cannot be expressed in $\lambda^{\triangleright}$, because there is no context C such that $x : \triangleright_\alpha b @\varepsilon \vdash^\beta C[x] : \triangleright_\alpha b$. A second difference is the behavior of **run** for the term $M : \forall\alpha.\triangleright_\alpha\triangleright_\alpha b$. In λ^α, **run** will remove only one quotation, leaving $\forall$, so **run** $M : \forall\alpha.\triangleright_\alpha b$, while, in $\lambda^{\triangleright}$, the application to ε removes all $\triangleright_\alpha$, that is, $M\,\varepsilon : b$.

More recently, Yuse and Igarashi have proposed the calculus $\lambda^{\bigcirc\Box}$ [10] by combining $\lambda^{\bigcirc}$ and $\lambda^{\Box}$, while maintaining the Curry-Howard isomorphism. The main idea was to consider LTL with modalities "always" ($\Box$) and "next" ($\bigcirc$), which represent closed and open code types, respectively. It is similar to AIM in this respect. Although $\lambda^{\bigcirc\Box}$ is based on logic, it cannot be embedded into $\lambda^{\triangleright}$ simply by combining the two embeddings above. In $\lambda^{\bigcirc\Box}$, both directions of $\Box\bigcirc\tau \leftrightarrow \bigcirc\Box\tau$ are provable, whereas neither direction of $(\forall\alpha.\triangleright_\alpha\triangleright_\beta\tau) \leftrightarrow \triangleright_\beta\forall\alpha.\triangleright_\alpha\tau$ is provable in $\lambda^{\triangleright}$. However, in $\lambda^{\bigcirc\Box}$ it seems impossible to program a code specializer like $\mathtt{power}_\forall$, which generates specialized code used at any stage; the best possible one presented can generate specialized code used only at any *later* stage, so running the specialized code is not possible.

It is considered not easy to develop a sound type system for staging constructs *with side effects*. Calcagno, Moggi, and Sheard developed a sound type system for a multi-stage calculus with references using closed types [18]. It is interesting to study whether their closedness condition can be relaxed by using α-closedness.

Other Multi-Stage Calculi. Calcagno, Yi, and Kim's λ^{poly}_{open} [11] is a rather powerful multi-stage calculus with open and closed code fragments, intentionally variable-capturing substitution, lifting values into code, and even references and ML-style type inference. The type structure of λ^{poly}_{open} is rather different: a code type records the names of free variables and their types, as well as the type of the whole code. It is not clear how (a pure fragment of) the calculus can be related to other foundational calculi; possible directions may be to use the calculus of contexts [19] by Sato, Sakurai, and Kameyama, and the contextual modal type theory by Nanevski, Pfenning, and Pientka [20].

Modal Logics. As we discussed above, the $\Box$-fragment of modal logic, the $\bigcirc$-fragment of LTL can be embedded into our logic, and the $\Box\bigcirc$-fragment of LTL and our logic cannot be comparable.

Our logic has three characteristic features: (1) it is multi-modal, (2) it has universal quantification over modalities and (3) modal operators are "relative", meaning their semantics depends on the possible world at which they are interpreted. Most of other logics do not have all of these features.

Dynamic logic [21] is a multi-modal logic for reasoning about programs. Its modal operators are $[\alpha]$ for each program α, and $[\alpha]\phi$ means "when α halts, ϕ must stand after execution of α from the current state". Dynamic logic is multi-modal and its modal operators are "relative", but does not have quantification over programs. Therefore, there is no formula in Dynamic logic which would correspond to $\forall\alpha.\, \triangleright_\alpha \triangleright_\alpha \phi$. There is, however, a formula which is expressive in Dynamic logic but not in our logic: e.g., a Dynamic logic formula $[\alpha^*]\phi$, which means intuitively $\phi \wedge [\alpha]\phi \wedge [\alpha][\alpha]\phi \wedge \dots$, cannot be expressed in our logic.

Hybrid logic [22] is a modal logic with a new kind of atomic formula called *nominals*, each of which must be true exactly one state in any model (therefore, a nonimal names a state). For each nominal i, $@_i$ is a modal operator and $@_i\phi$ means "ϕ stands at the state denoted by i". Hybrid logic has a universal quantifier over nominals. Hybrid logic differs from our logic, in that modal operators $@_i$ indicate worlds directly, hence are not "relative". In Hybrid logic $@_i @_j \phi \leftrightarrow @_j \phi$, but $\triangleright_\alpha \triangleright_\beta \phi$ and $\triangleright_\beta \phi$ are not equivalent in our logic.

6 Conclusion and Future Work

We have studied a logical aspect of environment classifiers by developing a simply typed multi-stage calculus $\lambda^\triangleright$ with environment classifiers. This calculus corresponds to a multi-modal logic with quantifier over transitions by the Curry-Howard isomorphism. The classical proof system is sound and complete with respect to the Kripke semantics. Our calculus simplifies the previous calculus λ^α of environment classifiers by reducing `run` and some use of CSP to an extension of another construct. We believe our work helps clarify the semantics of environment classifiers.

From a theoretical perspective, it is interesting to study the semantics of the intuitionistic version of the logic, as mentioned earlier, and also the calculus corresponding to the classical version of the logic. It is known that the naive combination of staging constructs and control operators is problematic since bound variables in quotation may escape from its scope by a control operator. We expect that a logical analysis, like the one presented here and Reed and Pfenning [23], will help analyze the problem.

From a practical perspective, one feature missing from $\lambda^\triangleright$ is CSP for all types. As argued in the introduction, we think typical use of CSP is rather limited and so easy to support. Type inference for $\lambda^\triangleright$ is an open problem, but, actually, Calcagno, Moggi, and Taha [15] have already developed type inference for a subset of λ^α, so it may be easy to apply their technique to $\lambda^\triangleright$.

Acknowledgments. This work was begun while the first author was at Kyoto University. We would like to thank Lintaro Ina, Naoki Kobayashi, Ryosuke Sato, and Naokata Shikuma for useful comments.

References

1. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993)
2. Consel, C., Lawall, J.L., Meur, A.F.L.: A tour of Tempo: A program specializer for the C language. Science of Computer Programming **52**(1-3) (2004) 341–370
3. Wickline, P., Lee, P., Pfenning, F.: Run-time code generation and Modal-ML. In: Proc. of PLDI'98 (1998) 224–235
4. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, M.F.: 'C and tcc: A language and compiler for dynamic code generation. ACM TOPLAS **21**(2) (1999) 324–369
5. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science **248** (2000) 211–242
6. Glück, R., Jørgensen, J.: Efficient multi-level generating extensions for program specialization. In: Proc. of PLILP'95. Volume 982 of LNCS. (1995) 259–278
7. Davies, R.: A temporal-logic approach to binding-time analysis. In: Proc. of LICS'96. (1996) 184–195
8. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM **48**(3) (2001) 555–604
9. Taha, W., Nielsen, M.F.: Environment classifiers. In: Proc. of POPL'03. (2003) 26–37
10. Yuse, Y., Igarashi, A.: A modal type system for multi-level generating extensions with persistent code. In: Proc. of PPDP'06. (2006) 201–212
11. Kim, I.S., Yi, K., Calcagno, C.: A polymorphic modal type system for lisp-like multi-staged languages. In: Proc. of POPL'06 (2006) 257–268
12. Stirling, C.: Modal and temporal logics. In: Handbook of Logic in Computer Science. Volume 2. Oxford University Press (1992) 477–563
13. Moggi, E., Taha, W., Benaissa, Z.E.A., Sheard, T.: An idealized MetaML: Simpler, and more expressive. In: Proc. of ESOP'99. Volume 1576 of LNCS. (1999) 193–207
14. Benaissa, Z.E.A., Moggi, E., Taha, W., Sheard, T.: Logical modalities and multi-stage programming. In: Proc. of IMLA'99. (1999)
15. Calcagno, C., Moggi, E., Taha, W.: ML-like inference for classifiers. In: Proc. of ESOP'04, Volume 2986 of LNCS. (2004) 79–93
16. Kojima, K., Igarashi, A.: On constructive linear-time temporal logic. In: Proc. of IMLA'08 (2008)
17. Nanevski, A., Pfenning, F.: Staged computation with names and necessity. J. Functional Programming **15**(5) (2005) 893–939
18. Calcagno, C., Moggi, E., Sheard, T.: Closed types for a safe imperative MetaML. Journal of Functional Programming **13**(3) (2003) 545–571
19. Sato, M., Sakurai, T., Kameyama, Y.: A simply typed context calculus with first-class environments. J. Functional and Logic Programming **2002**(4) (2002) 1–41
20. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic **9**(3) (2008)
21. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In Gabbay, D., Guenther, F., eds.: Handbook of Philosophical Logic. Volume 4. 2nd edn. Springer-Verlag (2002) 99–218
22. Areces, C., ten Cate, B.: Hybrid logics. In Blackburn, P., Wolter, F., van Benthem, J., eds.: Handbook of Modal Logics. Elsevier (2007) 821–868
23. Reed, J., Pfenning, F.: Intuitionistic letcc via labelled deduction. In: Proc. of M4M'07. (2007)