# A Dependently Typed Multi-Stage Calculus*

Akira Kawata[1] and Atsushi Igarashi[2][0000−0002−5143−9764]

[1] Graduate School of Informatics, Kyoto University, Kyoto, Japan
akira@fos.kuis.kyoto-u.ac.jp
[2] igarashi@kuis.kyoto-u.ac.jp

**Abstract.** We develop yet another typed multi-stage calculus $\lambda^{\mathrm{MD}}$. It extends Hanada and Igarashi's $\lambda^{\triangleright\%}$ with dependent types. A multi-stage calculus enables us to generate and execute codes at runtime. It can improve the performance of programs by generating optimized codes for given inputs. Dependent types are types dependent on values. A vector with its length is a famous example of dependent types and it enables us to omit boundary checking. In this paper, we design $\lambda^{\mathrm{MD}}$ by introducing dependent type into $\lambda^{\triangleright\%}$. You can make more efficient programs from existing dependent typed programs with $\lambda^{\mathrm{MD}}$. $\lambda^{\mathrm{MD}}$ has a simple, substitution-based full-reduction semantics and enjoys basic properties of subject reduction, confluence, and strong normalization, and progress. It also includes an evaluation context which satisfies unique decomposition. The main technical points of this paper are how to deal with Cross Stage Persistence of multi-stage calculuses which allows using a value in quoted code in a dependent type system. Especially, the way of handling CSP in equivalence rules of dependent types wasn't clear. In this paper, we give reasonable equivalence rules to handle them.

**Keywords:** Multistage programming · Dependent type

181 words. The abstract should briefly summarize the contents of the paper in 150–250 words.

## 1 Introduction

### 1.1 Multi-Stage Calculus

**Introduction to Multi-Stage Calculus**

**Application to Performance Imporving**

### 1.2 Dependent Types

**Introduction to Dependent Types**

* Supported by organization x.

**Application to Omitting Boundary Checking**

### 1.3 Organization of the Paper

## 2 Informal Overview of $\lambda^{\mathrm{MD}}$

In this section, we check $\lambda^{\mathrm{MD}}$ informally after checking $\lambda^{\triangleright\%}$ and $\lambda^{\mathrm{LF}}$ which are the basis of $\lambda^{\mathrm{MD}}$.

### 2.1 $\lambda^{\triangleright\%}$

In $\lambda^{\triangleright\%}$, quote and unquote are written $\blacktriangleright_\alpha\ M$ and $\blacktriangleleft_\alpha\ M$, respectively. The type of $\blacktriangleright_\alpha\ M$ is $\triangleright_\alpha \tau$ when $M$ has $\tau$ type. The type of $\blacktriangleleft_\alpha\ M$ is $\tau$ when $M$ has $\triangleright_\alpha \tau$. Please notice that $\blacktriangleleft_\alpha\ M$ is well-typed, $M$ has a code type from the typing rule of $\lambda^{\triangleright\%}$. In addition to normal $\beta$-reduction, there is an reduction rule for unquoted quoted code.

$$\blacktriangleleft_\alpha\ (\blacktriangleright_\alpha\ M) \longrightarrow M$$

It means when a quoted code fragment is unquoted, it returns back to the original code fragment. This reduction is called $\longrightarrow_\Lambda$ in $\lambda^{\mathrm{MD}}$.

The subscription $\alpha$ is a transition variable which is called a stage variable in $\lambda^{\mathrm{MD}}$. It is used to show the thickness of quoting. For example, $\blacktriangleright_\alpha\ (\lambda x : \mathrm{Int}.x{+}10)$ is a fragment of code which becomes $(\lambda x : \mathrm{Int}.x + 10)$ after run it once and $\blacktriangleright_\alpha \blacktriangleright_\beta\ (\lambda x : \mathrm{Int}.x + 10)$ becomes $(\lambda x : \mathrm{Int}.x + 10)$ after run it twice.

A sequence of transition variables is a transition which is called a stage in $\lambda^{\mathrm{MD}}$. In $\lambda^{\triangleright\%}$, all judgement has a transition. Especially, terms without quoting exist at the empty transition which is represented by $\epsilon$. For example, $(\lambda x : \mathrm{Int}.x)\ (1 + 2)$ is at $\epsilon$ transition and $\blacktriangleright_\alpha\ (\lambda x : \mathrm{Int}.x)$ is at $\alpha$ transition.

There are abstraction for transition variables and application for transition abstraction in $\lambda^{\triangleright\%}$. They looks like $\Lambda\alpha.M$ and $M\ A$, respectively. A transition abstraction binds a transition variable in a term. For example, all $\alpha$ in $\Lambda\alpha.(\blacktriangleright_\alpha\ (\lambda x : \mathrm{Int}.x))$ are bound. It is only natural there is a rule for transition application in $\lambda^{\triangleright\%}$. The rule is following.

$$(\Lambda\alpha.M)\ A \longrightarrow M[\alpha \mapsto A]$$

For example, $\Lambda\alpha.(\blacktriangleright_\alpha\ (\lambda x : \mathrm{Int}.x))\ (\beta\gamma)$ reduces to $\blacktriangleright_{\beta\gamma}\ (\lambda x : \mathrm{Int}.x)$. Although we can apply any transition to a transition abstraction in $\lambda^{\triangleright\%}$, we restrict the transitions only to $\epsilon$ transition in $\lambda^{\mathrm{MD}}$ to simplify the system.

Another important rule about a transition variable is that a transition variable related symbol disappears when the empty transition is substituted to the transition variable. For example, $(\blacktriangleright_\epsilon\ (\lambda x : \mathrm{Int}.x))$ is equivalent to $(\lambda x : \mathrm{Int}.x)$. This rule is applied all transition variable related symbols: $\blacktriangleright_\alpha$, $\blacktriangleleft_\alpha$, and $\%_\alpha$.

The main purpose of transition abstractions and transition applications are to represent **run** operator. In multistage calculus, **run** is a very important operator.

It changes quoted code to the original code. **run** is realized with application of the empty transition $\epsilon$. For example, $\Lambda\alpha.(\blacktriangleright_\alpha\ (\lambda x : \mathrm{Int}.x))\ \epsilon$ becomes $\lambda x : \mathrm{Int}.x$ because $\blacktriangleright_\epsilon$ is disappeared.

CSP, cross-stage persistence, is an important feature of multistage calculus. It enable us to embed value at an outer transition into an inner transition . $\%$ is dedicated for CSP in $\lambda^{\triangleright\%}$. For example, $\lambda a : \mathrm{Int}.\Lambda\alpha.(\blacktriangleright_\alpha\ (\lambda x : \mathrm{Int}.x + \%_\alpha a))$

There is another important function called program residualization in $\lambda^{\triangleright\%}$. It means that a generated code can be dumped into a file. We can load the dumped file and run it. The difficulty is arise when program residualization is used with CSP. Transition variables are classified to two kinds in $\lambda^{\triangleright\%}$ in order to deal with this difficulty.

### 2.2 Dependent Types

### 2.3 Extending $\lambda^{\triangleright\%}$ with Dependent Types

Next, we develop $\lambda^{\mathrm{MD}}$ by extending $\lambda^{\triangleright\%}$ with dependent types.

## 3 Formal Definition of $\lambda^{\mathrm{MD}}$

In this section, we present $\lambda^{\mathrm{MD}}$ in detail. We will define syntax, full reduction, type system including type equivalence rules, values, and evaluation context which take stages into account. In the next section, we will discuss on properties of $\lambda^{\mathrm{MD}}$.

### 3.1 Syntax

$\lambda^{\mathrm{MD}}$ is defined by the following grammar.

| | |
|---|---|
| Type variables | $X, Y, Z$ |
| Constants | $c$ |
| Variables | $x, y, z$ |
| Stage variables | $\alpha, \beta, \gamma$ |
| Stage | $A, B, C$ |
| Kinds | $K, J, I, H, G ::= * \mid \Pi x : \tau.K$ |
| Types | $\tau, \sigma, \rho, \pi, \xi ::= X \mid \Pi x : \tau.\tau \mid \tau\ M \mid \triangleright_\alpha \tau \mid \forall\alpha.\tau$ |
| Terms | $M, N, L, O, P ::= c \mid x \mid \lambda x : \tau.M \mid M\ M \mid \blacktriangleright_\alpha\ M \mid \blacktriangleleft_\alpha\ M \mid \Lambda\alpha.M \mid M\ \epsilon \mid \%_\alpha M$ |
| Signature | $\Sigma ::= \phi \mid X :: K \mid c : \tau$ |
| Contexts | $\Gamma ::= \phi \mid \Gamma, x : \tau@A\ (x \notin \mathrm{FV}(\Gamma))$ |

The metavariables $X, Y,$ and $Z$ range over the type variables, $c$ ranges over constants, the metavariables $x, y,$ and, $z$ range over the variables. The metavariables $\alpha, \beta,$ and $\gamma$ range over the transition variables. A transition, denoted by $A$ and $B$, is a finite sequence of transition variables. $\epsilon$ is a symbol for an empty transition.

A kind is $*$ or $\Pi x : \tau.K$. Types of terms have $*$ kinds and dependent types have $\Pi$-kinds.

A type is a type variable which is declared in the signature, a dependent type, a dependent type applied a term, a code type, or an $\alpha$-closed type. A dependent type $\Pi x : \tau.\tau$ is a type depending on values such as a vector with its length. A function type in ordinary type systems is omitted because it is a specialized case of a dependent type. A code type $\triangleright_\alpha \tau$ denotes a code fragment of a term of type $\tau$. An $\alpha$-closed type corresponds to a runnable code fragment.

In addition to normal terms in Simply Typed Lambda Calculus, there are five more forms. $\blacktriangleright_\alpha M$ represents a code fragment, and $\blacktriangleright_\alpha M$ represents unquote. $\Lambda\alpha.M$ is a stage variable abstraction. $M\ \epsilon$ is an application of $\epsilon$ to a stage variable. $\%_\alpha M$ is a primitive operator for cross-stage persistence.

Signatures are sequences of pairs of a constant and its type or a type variable and its kind. Because we adopted a $\lambda^{\mathrm{LF}}$-like system, constants and type variables for base types are given in the signature $\Sigma$. For example, when we use integers in $\lambda^{\mathrm{MD}}$, $\Sigma = \mathrm{Int} :: *, 1 : \mathrm{Int}, 2 : \mathrm{Int}, \cdots$.

Contexts are sequences of triplexes of a variable, its type, and its stage. In order to restrict contexts to only well-formed ones, there is a condition of $x \notin \mathrm{FV}(\Gamma)$. Because of this condition, there is no type with free variables in contexts.

Connection degree of $\blacktriangleright_\alpha, \blacktriangleleft_\alpha, \%_\alpha$ is stronger than the two forms of applications and applications are left-associative and two abstractions extends as far to the right as possible.

As usual, the variable $x$ is bound in $\lambda x : \tau.M$ and the transition variable $\alpha$ is bound in $\Lambda\alpha.M$. We identify $\alpha$-convertible terms and assume the names of bound variables are pairwise distinct. We write $\mathrm{FV}(M)$ and $\mathrm{FSV}(M)$ for the set of free variables and the set of free stage variables in $M$, respectively. We omit their definitions.

### 3.2 Reduction

In this section, we define full reduction for $\lambda^{\mathrm{MD}}$. Before giving the definition of reduction, we define two substitutions. Substitution $M[x \mapsto N]$ is the normal capture-avoiding substitution, and we omit its definition here. Substitution $M[\alpha \mapsto \epsilon]$ is defined below.

$$
\begin{aligned}
(\lambda x : \tau.M)[\alpha \mapsto \epsilon] &= \lambda x : \tau[\alpha \mapsto \epsilon].M[\alpha \mapsto \epsilon] \\
(M\ N)[\alpha \mapsto \epsilon] &= (M[\alpha \mapsto \epsilon])\ (N[\alpha \mapsto \epsilon]) \\
(\blacktriangleright_\beta\ M)[\alpha \mapsto \epsilon] &= \blacktriangleright_{\beta[\alpha \mapsto \epsilon]}\ M[\alpha \mapsto \epsilon] \\
(\blacktriangleleft_\beta\ M)[\alpha \mapsto \epsilon] &= \blacktriangleleft_{\beta[\alpha \mapsto \epsilon]}\ M[\alpha \mapsto \epsilon] \\
(\Lambda\beta.M)[\alpha \mapsto \epsilon] &= \Lambda\beta.M[\alpha \mapsto \epsilon] && (\text{if } \alpha \neq \beta) \\
(\Lambda\beta.M)[\alpha \mapsto \epsilon] &= \Lambda\beta.M && (\text{if } \alpha = \beta) \\
(M\ \epsilon)[\alpha \mapsto \epsilon] &= M[\alpha \mapsto \epsilon]\ \epsilon \\
(\%_\beta M)[\alpha \mapsto \epsilon] &= \%_{\beta[\alpha \mapsto \epsilon]} M[\alpha \mapsto \epsilon]
\end{aligned}
$$

**Definition 1 (Reduction).** *There are three reduction rules ($\longrightarrow_\beta, \longrightarrow_\blacklozenge, \longrightarrow_\Lambda$) in $\lambda^{MD}$. Congruence rules which are omitted from the definition.*

$$
\begin{aligned}
&(\lambda x : \tau.M)N \longrightarrow_\beta M[x \mapsto N] \\
&\blacktriangleleft_\alpha \blacktriangleright_\alpha\ M \longrightarrow_\blacklozenge M \\
&(\Lambda\alpha.M)\ \epsilon \longrightarrow_\Lambda M[\alpha \mapsto \epsilon]
\end{aligned}
$$

*We write $M \longrightarrow M'$ iff $M \longrightarrow_\beta M'$, $M \longrightarrow_\blacklozenge M'$, or $M \longrightarrow_\Lambda M'$.*

$\longrightarrow_\beta$ is normal $\beta-$reduction in lambda calculus. $\longrightarrow_\blacklozenge$ means that when a quoted code is unquoted, it becomes the original code. $\longrightarrow_\Lambda$ means that a stage abstraction applied to the empty stage $\epsilon$ reduces to the body of abstraction where $\epsilon$ is substituted to the stage variable. Application of a non-$\epsilon$ stage to a stage abstraction is prohibited in order to simplify $\lambda^{\mathrm{MD}}$. There is no reduction rule for CSP as with Hanada and Igarashi [4]. $\%_\alpha$, the symbol for CSP, is disappeared when $\epsilon$ is substituted to $\alpha$. $\blacktriangleright_\alpha$ and $\blacktriangleleft_\alpha$ is disappeared in the same way as $\%_\alpha$.

### 3.3 Type System

In this section, we define the type system of $\lambda^{\mathrm{MD}}$. $\lambda^{\mathrm{MD}}$ is little complicated because it contains dependent types. It contains typing, kinding, well-formed kinding, term equality, type equality, and kind equality. In other words, there are six types of judgement in $\lambda^{\mathrm{MD}}$. We show them in Figure 1. Especially, we discuss on the type equality of $\lambda^{\mathrm{MD}}$ in detail because it is the main contribution of this paper. $\lambda^{\mathrm{MD}}$ is developed on the basis of $\lambda^{\triangleright\%}$ but there are no type equivalence rules in $\lambda^{\triangleright\%}$.

**Definition 2 (Typing).** *The typing relation $\Gamma \vdash_\Sigma M : \tau@A$ is the least relation closed under the rules in Figure 3.*

We show typing rules of $\lambda^{\mathrm{MD}}$ in Figure 3. The rules T-Var, T-Abs, and T-App are almost the same as those in simply typed lambda calculus if you ignore stage annotations and kind checking. The rule T-Var means that a variable can appear only at the stage in which it is declared. It checks the type of variable $x$, $\tau$, has the proper kind for terms, $*$. T-Abs also checks the kind of the type.

$$\Gamma \vdash_\Sigma M : \tau @ A$$
$$\Gamma \vdash_\Sigma \tau :: K @ A$$
$$\Gamma \vdash_\Sigma K \text{ kind} @ A$$
$$\Gamma \vdash_\Sigma M \equiv N @ A$$
$$\Gamma \vdash_\Sigma \tau \equiv \sigma$$
$$\Gamma \vdash_\Sigma K \equiv J @ A$$

**Fig. 1.** Six types of judgement in $\lambda^{\mathrm{MD}}$

The rule T-Const means any constants in the signature can appear at any stage. For example, if we have a signature $\Sigma$ which is bool :: $*$, true : bool, false : bool, the derivation tree in Figure 2 is admissible.

$$\frac{\text{true} : \text{bool} \in \Sigma \qquad \dfrac{\vdots}{\Gamma \vdash_\Sigma \text{bool} :: * @ \alpha\beta}}{\Gamma \vdash_\Sigma \text{true} : \text{bool} @ \alpha\beta}\ \text{T-Const}$$

**Fig. 2.** A derivation tree using T-Const

The rules T-▶, T-◀, T-Gen, T-Ins, and T-Csp are rules for a multistage calculus. They are corresponding to quoting a code, unquoting a code, making a stage abstraction, application of $\epsilon$ stage to a stage abstraction, and cross-stage persistence, respectively. Please check Hanada and Igarashi [4] and Tsukada and Igarashi [16] for details of these rules.

The main technical point of this paper is T-Conv and type equivalence rules needed by T-Conv. This rule allows ut to replace a type with another type that is equivalent. In a type system which includes dependent types, this kind of rule is essential because two types which have different shapes may be equivalent. For example, when we use a matrix type which have their sizes (Mat $n$ $m$), Mat 5 3 is equivalent to Mat $(4+1)$ $(1+2)$ ~~obviously~~. obvious?

**Type Equivalence Rules**

As mentioned above, type equivalence rules are needed in $\lambda^{\mathrm{MD}}$. We show the rules in Figure 4.

When we design type equivalence rules of $\lambda^{\mathrm{MD}}$, there are two design choices. One is defining type equivalence by $\beta$-equality after we define $\beta$-reduction of types. Another is defining type equivalence directly by a combination of type equivalence rules. We adopt the latter one because it is convenient to handle CSP in the type equivalence.

このちがいが何がどうちがいの結果かピンとこない

$$\frac{c : \tau \in \Sigma \qquad \Gamma \vdash_\Sigma \tau :: *@A}{\Gamma \vdash_\Sigma c : \tau@A}\text{T-Const} \qquad \frac{x : \tau@A \in \Gamma \qquad \Gamma \vdash_\Sigma \tau :: *@A}{\Gamma \vdash_\Sigma x : \tau@A}\text{T-Var}$$

$$\frac{\Gamma \vdash_\Sigma \sigma :: *@A \qquad \Gamma, x : \sigma@A \vdash_\Sigma M : \tau@A}{\Gamma \vdash_\Sigma (\lambda(x : \sigma).M) : (\Pi(x : \sigma).\tau)@A}\text{T-Abs}$$

$$\frac{\Gamma \vdash_\Sigma M : (\Pi(x : \sigma).\tau)@A \qquad \Gamma \vdash_\Sigma N : \sigma@A}{\Gamma \vdash_\Sigma M\ N : \tau[x \mapsto N]@A}\text{T-App}$$

$$\frac{\Gamma \vdash_\Sigma M : \tau@A \qquad \Gamma \vdash_\Sigma \tau \equiv \sigma :: K@A}{\Gamma \vdash_\Sigma M : \sigma@A}\text{T-Conv}$$

$$\frac{\Gamma \vdash_\Sigma M : \tau@A\alpha}{\Gamma \vdash_\Sigma \blacktriangleright_\alpha M : \triangleright_\alpha \tau@A}\text{T-}\blacktriangleright \qquad \frac{\Gamma \vdash_\Sigma M : \triangleright_\alpha \tau@A}{\Gamma \vdash_\Sigma \blacktriangleleft_\alpha M : \tau@A\alpha}\text{T-}\blacktriangleleft$$

$$\frac{\Gamma \vdash_\Sigma M : \tau@A \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\mathrm{A})}{\Gamma \vdash_\Sigma \Lambda\alpha.M : \forall\alpha.\tau@A}\text{T-Gen}$$

$$\frac{\Gamma \vdash_\Sigma M : \forall\alpha.\tau@A}{\Gamma \vdash_\Sigma M\ \epsilon : \tau[\alpha \mapsto \epsilon]@A}\text{T-Ins} \qquad \frac{\Gamma \vdash_\Sigma M : \tau@A}{\Gamma \vdash_\Sigma \%_\alpha M : \tau@A\alpha}\text{T-Csp}$$

**Fig. 3.** Typing Rules

We show type equivalence rules in Figure 4. All rules except QT-Refl, QT-Sym, QT-Trans, and QT-App are generated naturally from the typing rules. QT-Refl, QT-Sym, QT-Trans exist in order to make the type equivalence relation an equivalence relationship. The rule QT-App means that if there are two equivalent $\Pi$ type and two equivalent terms, the results of applications are also equivalent. A judgement of $\Gamma \vdash_\Sigma M \equiv N : \rho@A$ means that $M$ and $N$ are equivalent.

The equivalence of terms is defined in Figure 5. Q-Abs, Q-App, Q-$\blacktriangleright$, Q-$\blacktriangleleft$, Q-Gen, Q-Ins, Q-Csp are generated directly from the syntax of a term. Q-Refl, Q-Sym, Q-Trans exist in order to make the term equivalence relation an equivalence relationship as with the type equivalence. The rule Q-$\beta$ means when a term $M \longrightarrow_\beta M'$, $M$ is equivalent to $M'$. Q-$\blacktriangleleft\blacktriangleright$and Q-$\Lambda$ are corresponding to $\longrightarrow_\blacklozenge$ and $\longrightarrow_\Lambda$, respectively.

The rule Q-% is something special because it has no background of syntax or reductions. But we add this rule to $\lambda^{\mathrm{MD}}$ because it is essential when you use dependent types in code type. For example, we can utilize a dependently typed multistage calculus to generate code for matrix multiplication. When we know sizes of matrices before the multiplication, we can generate efficient code for multiplication by unrolling loops.

mulmat function in the following code fragment generate code for matrix multiplication. mulmat takes two integers which are the size of the multiplicand matrix and generate a code. The last integer to decide the size of the multiplier matrix is given at runtime. You can generate code by applying two integers to mulmat. We applied 3 and 5 in the second line and got $\triangleright_\alpha \Pi z$ : Int.(Mat $z$ $\%_\alpha 5$) $\rightarrow$ (Mat $\%_\alpha 5$ $\%_\alpha 3$) $\rightarrow$ (Mat $z$ $\%_\alpha 3$). But this type is difficult

$$\frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: *@A \qquad \Gamma, x : \tau@A \vdash_\Sigma \rho \equiv \pi :: *@A}{\Gamma \vdash_\Sigma \Pi x : \tau.\rho \equiv \Pi x : \sigma.\pi :: *@A}\text{QT-Abs}$$

$$\frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: (\Pi x : \rho.K)@A \qquad \Gamma \vdash_\Sigma M \equiv N : \rho@A}{\Gamma \vdash_\Sigma \tau\ M \equiv \sigma\ N :: K[x \mapsto M]@A}\text{QT-App}$$

$$\frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: *@A\alpha}{\Gamma \vdash_\Sigma \triangleright_\alpha \tau \equiv \triangleright_\alpha \sigma :: *@A}\text{QT-}\triangleright \qquad \frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: K@A}{\Gamma \vdash_\Sigma \tau \equiv \sigma :: K@A\alpha}\text{QT-Csp}$$

$$\frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: *@A \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\mathrm{A})}{\Gamma \vdash_\Sigma \forall\alpha.\tau \equiv \forall\alpha.\sigma :: *@A}\text{QT-Gen}$$

$$\frac{\Gamma \vdash_\Sigma \tau :: K@A}{\Gamma \vdash_\Sigma \tau \equiv \tau :: K@A}\text{QT-Refl} \qquad \frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: K@A}{\Gamma \vdash_\Sigma \sigma \equiv \tau :: K@A}\text{QT-Sym}$$

$$\frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: K@A \qquad \Gamma \vdash_\Sigma \sigma \equiv \rho :: K@A}{\Gamma \vdash_\Sigma \tau \equiv \rho :: K@A}\text{QT-Trans}$$

**Fig. 4.** Type Equivalence Rules

to combine with other code because there are two $\%_\alpha$ for CSP. Q-% states that we can erase this $\%_\alpha$ under a condition which is explained in the next paragraph.

The condition is that a CSPed value equals to the original value when it has the same type in the original stage. For example, $\vdash_\Sigma \%_\alpha 5 \equiv 5@\alpha$ because $\vdash_\Sigma 5 : \mathrm{Int}@\alpha$ from T-Const and $\vdash_\Sigma \%_\alpha 5 : \mathrm{Int}@\alpha$. In other words, we can remove a $\%_\alpha$ symbol of a value when it doesn't change the type.

$$\begin{aligned}&\mathrm{mulmat} : \Pi x : \mathrm{Int}.\Pi y : \mathrm{Int}.(\triangleright_\alpha \Pi z : \mathrm{Int}.(\mathrm{Mat}\ z\ \%_\alpha y) \to (\mathrm{Mat}\ \%_\alpha y\ \%_\alpha x) \to (\mathrm{Mat}\ z\ \%_\alpha x))\\ &\mathrm{mulmat}\ 3\ 5 : \triangleright_\alpha \Pi z : \mathrm{Int}.(\mathrm{Mat}\ z\ \%_\alpha 5) \to (\mathrm{Mat}\ \%_\alpha 5\ \%_\alpha 3) \to (\mathrm{Mat}\ z\ \%_\alpha 3)\\ &\qquad (\equiv \triangleright_\alpha \Pi z : \mathrm{Int}.(\mathrm{Mat}\ z\ 5) \to (\mathrm{Mat}\ 5\ 3) \to (\mathrm{Mat}\ z\ 3))\end{aligned}$$

This kind of type equality is very useful when we combine mulmat 3 5 with another code. We cannot remove $\%_\alpha$ in the type without Q-%and it makes very difficult to combine because types of codes don't contain $\%_\alpha$ generally.

The reduction given above is a full reduction and any redexes can be reduced in arbitrary order. But we need to fix order of reduction when we implement an interpreter of $\lambda^{\mathrm{MD}}$ and the order must be appropriate in terms of stages.

We will define deterministic call-by-value staged semantics which can be used as a basis of a $\lambda^{\mathrm{MD}}$ interpreter. In this reduction, $\longrightarrow_\beta$ and $\longrightarrow_\Lambda$ are allowed only at stage $\epsilon$ and $\longrightarrow_\blacklozenge$ is allowed only at stage $\alpha$.

**Definition 3 (Values).** *The family $V^A$ of sets of values, ranged over by $v^A$ and the sets of $\epsilon$-redexes (ranged over by $R^\epsilon$) and $\alpha$-redexes (ranged over by $R^\alpha$)*

$$\frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: *@A \qquad \Gamma, x : \tau@A \vdash_\Sigma M \equiv N : \rho@A}{\Gamma \vdash_\Sigma \lambda x : \tau.M \equiv \lambda x : \sigma.N : (\Pi x : \tau.\rho)@A}\text{Q-Abs}$$

$$\frac{\Gamma \vdash_\Sigma M \equiv L : (\Pi x : \sigma.\tau)@A \qquad \Gamma \vdash_\Sigma N \equiv O : \sigma@A}{\Gamma \vdash_\Sigma M\ N \equiv L\ O : \tau[x \mapsto N]@A}\text{Q-App}$$

$$\frac{\Gamma \vdash_\Sigma M \equiv N : \tau@A\alpha}{\Gamma \vdash_\Sigma \blacktriangleright_\alpha M \equiv \blacktriangleright_\alpha N : \triangleright_\alpha \tau@A}\text{Q-}\blacktriangleright \qquad \frac{\Gamma \vdash_\Sigma M \equiv N : \triangleright_\alpha \tau@A}{\Gamma \vdash_\Sigma \blacktriangleleft_\alpha M \equiv \blacktriangleleft_\alpha N : \tau@A\alpha}\text{Q-}\blacktriangleleft$$

$$\frac{\Gamma \vdash_\Sigma M \equiv N : \tau@A \qquad \alpha \notin \mathrm{FSV}(\Gamma) \cup \mathrm{FSV}(A)}{\Gamma \vdash_\Sigma \Lambda\alpha.M \equiv \Lambda\alpha.N : \forall\alpha.\tau@A}\text{Q-Gen}$$

$$\frac{\Gamma \vdash_\Sigma M \equiv N : \forall\alpha.\tau@A}{\Gamma \vdash_\Sigma M\ \epsilon \equiv N\ \epsilon : \tau[\alpha \mapsto \epsilon]@A}\text{Q-Ins} \qquad \frac{\Gamma \vdash_\Sigma M \equiv N : \tau@A}{\Gamma \vdash_\Sigma \%_\alpha M \equiv \%_\alpha N : \tau@A\alpha}\text{Q-Csp}$$

$$\frac{\Gamma \vdash_\Sigma M : \tau@A}{\Gamma \vdash_\Sigma M \equiv M : \tau@A}\text{Q-Refl} \qquad \frac{\Gamma \vdash_\Sigma M \equiv N : \tau@A}{\Gamma \vdash_\Sigma N \equiv M : \tau@A}\text{Q-Sym}$$

$$\frac{\Gamma \vdash_\Sigma M \equiv N : \tau@A \qquad \Gamma \vdash_\Sigma N \equiv L : \tau@A}{\Gamma \vdash_\Sigma M \equiv L : \tau@A}\text{Q-Trans}$$

$$\frac{\Gamma, x : \sigma@A \vdash_\Sigma M : \tau@A \qquad \Gamma \vdash_\Sigma N : \sigma@A}{\Gamma \vdash_\Sigma (\lambda x : \sigma.M)\ N \equiv M[x \mapsto N] : \tau[x \mapsto N]@A}\text{Q-}\beta$$

$$\frac{\Gamma \vdash_\Sigma M \equiv N : \tau@A}{\Gamma \vdash_\Sigma \blacktriangleleft_\alpha (\blacktriangleright_\alpha M) \equiv N : \tau@A}\text{Q-}\blacktriangleleft\blacktriangleright$$

$$\frac{\Gamma \vdash_\Sigma (\Lambda\alpha.M) : \forall\alpha.\tau@A}{\Gamma \vdash_\Sigma (\Lambda\alpha.M)\ \epsilon \equiv M[\alpha \mapsto \epsilon] : \tau[\alpha \mapsto \epsilon]@A}\text{Q-}\Lambda$$

$$\frac{\Gamma \vdash_\Sigma M : \tau@A\alpha \qquad \Gamma \vdash_\Sigma M : \tau@A}{\Gamma \vdash_\Sigma \%_\alpha M \equiv M : \tau@A\alpha}\text{Q-}\%$$

**Fig. 5.** Tem Equivalence Rules

*are defined by following grammar. In the grammar* $A \neq \epsilon$.

$$
\begin{array}{lll}
\textit{Values} & v^{\epsilon} \in V^{\epsilon} ::= \lambda x : \tau.M \mid \blacktriangleright_{\alpha}\ v^{\alpha} \mid \Lambda\alpha.v^{\epsilon} \\
& v^{A} \in V^{A} ::= x \mid \lambda x : \tau.v^{A} \mid v^{A}\ v^{A} \mid \blacktriangleright_{\alpha}\ v^{A\alpha} \mid \Lambda\alpha.v^{A} \mid v^{A}\ \epsilon \\
& \qquad\qquad \mid \blacktriangleleft_{\alpha}\ v^{A'} (\textit{if } A'\alpha = A \textit{ and } A' \neq \epsilon) \\
& \qquad\qquad \mid \%_{\alpha} v^{A'} (\textit{if } A'\alpha = A) \\
\textit{Redexes} & R^{\epsilon} ::= (\lambda x : \tau.M)\ v^{\epsilon} \mid (\Lambda\alpha.v^{\epsilon})\ \epsilon \\
& R^{\alpha} ::= \blacktriangleleft_{\alpha}\blacktriangleright_{\alpha}\ M
\end{array}
$$

Values at $\epsilon$ stage are a $\lambda$ abstraction, a quoted code, or a $\Lambda$ abstraction. The body of a $\lambda$ abstraction can be any term but the body of $\Lambda$ abstraction only can be a value. This restriction means that the body of *Lambda* abstraction must be evaluated before application of an $\epsilon$ stage. Values at non-$\epsilon$ stage includes all term except $\blacktriangleleft_{\alpha}\ v^{\epsilon}$. This is because $\blacktriangleleft_{\alpha}\ v^{\epsilon}$ is a redex.

**Definition 4 (Evaluation Context).** *The family of sets* $ECtx^{A}_{B}$ *of evaluation contexts, ranged over by* $E^{A}_{B}$*, is defined by the following grammar,* $A$ *is nonempty. (Although* $B, A'$ *can be empty.)* $A \neq \epsilon$

$$
\begin{array}{l}
E^{\epsilon}_{B} \in ECtx^{\epsilon}_{B} ::= \Box\ (\textit{if } B = \epsilon) \mid E^{\epsilon}_{B}\ M \mid v^{e}\ E^{\epsilon}_{B} \mid \blacktriangleright_{\alpha}\ E^{\alpha}_{B} \mid \Lambda\alpha.E^{\epsilon}_{B} \mid E^{\epsilon}_{B}\ \epsilon \\
E^{A}_{B} \in ECtx^{A}_{B} ::= \Box\ (\textit{if } A = B) \mid \lambda x : \tau.E^{A}_{B} \mid E^{A}_{B}\ M \mid v^{A}\ E^{A}_{B} \mid E^{\epsilon}_{B} \mid \blacktriangleright_{\alpha}\ E^{A\alpha}_{B} \mid \blacktriangleleft_{\alpha}\ E^{A'}_{B}\ (\textit{where } A'\alpha = A) \\
\qquad\qquad \mid \Lambda\alpha.E^{\epsilon}_{B} \mid E^{A}_{B}\ \epsilon \mid \%_{\alpha}\ E^{A'}_{B}\ (\textit{where } A'\alpha = A)
\end{array}
$$

We write $E^{A}_{B}[M]$ for a term obtained by filling the hole in $E^{A}_{B}$ with $M$.

**Definition 5 (Staged Reduction).** *The staged reduction relation, written* $M \longrightarrow_{s} M'$*, is defined by the least relation closed under the rules in Figure 6.*

$$
\begin{array}{c}
E^{A}_{\epsilon}[(\lambda x : \tau.M)\ v^{\epsilon}] \longrightarrow_{s} E^{A}_{\epsilon}[M[x \mapsto v^{\epsilon}]] \\
E^{A}_{\epsilon}[(\Lambda\alpha.v^{\epsilon})\ \epsilon] \longrightarrow_{s} E^{A}_{\epsilon}[v^{\epsilon}[\alpha \mapsto \epsilon]] \\
E^{A}_{\alpha}[\blacktriangleleft_{\alpha}\blacktriangleright_{\alpha}\ v^{\alpha}] \longrightarrow_{s} E^{A}_{\alpha}[v^{\alpha}]
\end{array}
$$

**Fig. 6.** Staged Reduction

## 4 Properties of $\boldsymbol{\lambda^{\mathrm{MD}}}$

In this section, we prove some properties of $\lambda^{\mathrm{MD}}$: subject reduction, strong normalization, confluence, unique decomposition of evaluation contexts, and progress.

Substitution lemma for $\lambda^{\mathrm{MD}}$ is little more complicated than an ordinary one because there are six types of judgment and two types of substitution in $\lambda^{\mathrm{MD}}$.

**Theorem 1 (Substituition Lemma).**

$$\begin{aligned}
&\textit{If } \Gamma, z : \xi@B \vdash_\Sigma M : \tau@A \textit{ and } \Gamma \vdash_\Sigma P : \xi@B \textit{ then } \Gamma \vdash_\Sigma M[z \mapsto P] : \tau[z \mapsto P]@A.\\
&\textit{If } \Gamma, z : \xi@B \vdash_\Sigma \tau :: K@A \textit{ and } \Gamma \vdash_\Sigma P : \xi@B \textit{ then } \Gamma \vdash_\Sigma \tau[z \mapsto P] :: K[z \mapsto P]@A.\\
&\textit{If } \Gamma, z : \xi@B \vdash_\Sigma K \textit{ kind}@A \textit{ and } \Gamma \vdash_\Sigma P : \xi@B \textit{ then } \Gamma \vdash_\Sigma K[z \mapsto P] \textit{ kind}@A.\\
&\textit{If } \Gamma, z : \xi@B \vdash_\Sigma M \equiv N : \tau@A \textit{ and } \Gamma \vdash_\Sigma P : \xi@B \textit{ then } \Gamma \vdash_\Sigma M[z \mapsto P] \equiv N[z \mapsto P] : \tau[z \mapsto P]@A.\\
&\textit{If } \Gamma, z : \xi@B \vdash_\Sigma \tau \equiv \sigma : K@A \textit{ and } \Gamma \vdash_\Sigma P : \xi@B \textit{ then } \Gamma \vdash_\Sigma \tau[z \mapsto P] \equiv \sigma[z \mapsto P] : K[z \mapsto P]@A.\\
&\textit{If } \Gamma, z : \xi@B \vdash_\Sigma K \equiv J@A \textit{ and } \Gamma \vdash_\Sigma P : \xi@B \textit{ then } \Gamma \vdash_\Sigma K[z \mapsto P] \equiv J[z \mapsto P]@A.
\end{aligned}$$

*Proof.* Straightforward induction on derivations.

We need to prove another substitution lemma because there are stage variables.

**Theorem 2 (Stage Substituition Lemma).**

$$\begin{aligned}
&\textit{If } \Gamma \vdash_\Sigma M : \tau@A \textit{ then } \Gamma[\beta \mapsto \epsilon] \vdash_\Sigma M[\beta \mapsto \epsilon] : \tau[\beta \mapsto \epsilon]@A[\beta \mapsto \epsilon].\\
&\textit{If } \Gamma \vdash_\Sigma \tau :: K@A \textit{ then } \Gamma[\beta \mapsto \epsilon] \vdash_\Sigma \tau[\beta \mapsto \epsilon] :: K[\beta \mapsto \epsilon]@A[\beta \mapsto \epsilon].\\
&\textit{If } \Gamma \vdash_\Sigma K \textit{ kind}@A \textit{ then } \Gamma[\beta \mapsto \epsilon] \vdash_\Sigma K[\beta \mapsto \epsilon] \textit{ kind}@A[\beta \mapsto \epsilon].\\
&\textit{If } \Gamma \vdash_\Sigma M \equiv N : \tau@A \textit{ then } \Gamma[\beta \mapsto \epsilon] \vdash_\Sigma M[\beta \mapsto \epsilon] \equiv N[\beta \mapsto \epsilon] : \tau[\beta \mapsto \epsilon]@A[\beta \mapsto \epsilon].\\
&\textit{If } \Gamma \vdash_\Sigma \tau \equiv \sigma : K@A \textit{ then } \Gamma[\beta \mapsto \epsilon] \vdash_\Sigma \tau[\beta \mapsto \epsilon] \equiv \sigma[\beta \mapsto \epsilon] : K[\beta \mapsto \epsilon]@A[\beta \mapsto \epsilon].\\
&\textit{If } \Gamma \vdash_\Sigma K \equiv J@A \textit{ then } \Gamma[\beta \mapsto \epsilon] \vdash_\Sigma K[\beta \mapsto \epsilon] \equiv J[\beta \mapsto \epsilon]@A[\beta \mapsto \epsilon].
\end{aligned}$$

*Proof.* Straightforward induction on derivations.

Following three inversion lemmas are needed in proving later theorems.

**Lemma 1 (Inversion Lemma for $\Pi$ type).** *If* $\Gamma \vdash_\Sigma (\lambda x : \sigma.M) : (\Pi x : \sigma'.\tau)@A$ *then*

1. $\Gamma \vdash_\Sigma \sigma \equiv \sigma'@A$
2. $\Gamma, x : \sigma@A \vdash_\Sigma M : \tau@A$

*If* $\Gamma \vdash_\Sigma \rho \equiv (\Pi x : \sigma.\tau) : K@A$ *then* $\exists \sigma', \tau', K, J$ *such that*

1. $\rho = \Pi x : \sigma'.\tau'$
2. $\Gamma \vdash_\Sigma \sigma \equiv \sigma' : K@A$
3. $\Gamma, x : \sigma@A \vdash_\Sigma \tau \equiv \tau' : J@A$

*If* $\Gamma \vdash_\Sigma (\Pi x : \sigma.\tau) \equiv \rho : K@A$ *then* $\exists \sigma', \tau', K, J$ *such that*

1. $\rho = \Pi x : \sigma'.\tau'$
2. $\Gamma \vdash_\Sigma \sigma \equiv \sigma' : K@A$
3. $\Gamma, x : \sigma@A \vdash_\Sigma \tau \equiv \tau' : J@A$

*Proof.* Straightforward induction on derivations.

**Theorem 3 (Inversion Lemma for $\triangleright$ type).**

*If* $\Gamma \vdash_\Sigma \blacktriangleright_\alpha M : \triangleright_\alpha \tau@A$ *then* $\Gamma \vdash_\Sigma M : \tau@A$.
*If* $\Gamma \vdash_\Sigma \rho \equiv \triangleright_\alpha \tau : K@A$ *then* $\exists \tau', K, J$ *such that*

1. $\rho = \triangleright_\alpha \tau'$
2. $\Gamma@A \vdash_\Sigma \tau \equiv \tau' : K@A$

*If* $\Gamma \vdash_\Sigma \triangleright_\alpha \tau \equiv \rho : K@A$ *then* $\exists \tau', K, J$ *such that*

1. $\rho = \triangleright_\alpha \tau'$
2. $\Gamma@A \vdash_\Sigma \tau \equiv \tau' : K@A$

*Proof.* Straightforward induction on derivations.

**Theorem 4 (Inversion Lemma for $\Lambda$ type).**

*If* $\Gamma \vdash_\Sigma \Lambda\alpha.M : \forall\alpha.\tau@A$ *then* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $\alpha \notin FSV(\Gamma) \cup FV(A)$.
*If* $\Gamma \vdash_\Sigma \rho \equiv \forall\alpha.\tau : K@A$ *then* $\exists \tau', K$ *such that*

1. $\rho = \forall\alpha.\tau'$
2. $\Gamma \vdash_\Sigma \tau \equiv \tau' : K@A$

*If* $\Gamma \vdash_\Sigma \forall\alpha.\tau \equiv \rho : K@A$ *then* $\exists \tau', K$ *such that*

1. $\rho = \forall\alpha.\tau'$
2. $\Gamma \vdash_\Sigma \tau \equiv \tau' : K@A$

*Proof.* Straightforward induction on derivations.

Reductions preserve typing.

**Theorem 5 (Preservation).** *If* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $M \longrightarrow M'$, *then* $\Gamma \vdash_\Sigma M' : \tau@A$

*Proof.* First, there are three cases for $M \longrightarrow M'$. They are $M \longrightarrow_\beta M'$, $M \longrightarrow_\Lambda M'$, and $M \longrightarrow_\blacklozenge M'$. For each case, we can use straightforward induction on derivations. Difficult cases are T-App, T-$\blacktriangleleft$, and T-Ins. We need Inversion Lemmas for them.

No typed term has an infinite reduction sequence.

**Theorem 6 (Strong Normalization).** *If $\Gamma \vdash^{A}_{\Sigma} M : \tau$ then there is no infinite sequence of terms $(M_i)_{i\geq 1}$ and $M_i \longrightarrow M_{i+1}$ for $i \geq 1$*

*Proof.* In order prove this theorem, we define a translation $\natural$ from $\lambda^{\mathrm{MD}}$ to Simply Typed Lambda Calculus. Second, we prove $\natural$ preserves typing and $\beta$-reductions. Then, we can prove Strong Normalization of $\lambda^{\mathrm{MD}}$ from Strong Normalization of Simply Typed Lambda Calculus.

Any reduction sequences from one typed term converge. Because we have proved Strong Normalization, we can use Newman's Lemma to prove Confluence.

**Theorem 7 (Confluence).** *For any term $M$, if $M \longrightarrow^{*} M'$ and $M \longrightarrow^{*} M''$ then there exists $M'''$ that satisfies $M' \longrightarrow^{*} M'''$ and $M'' \longrightarrow^{*} M'''$.*

*Proof.* Because we proved Strong Normalization of $\lambda^{\mathrm{MD}}$, we can use Newman's lemma to prove Confluence of $\lambda^{\mathrm{MD}}$. Then, what we must show is Weak Church-Rosser Property now. When we consider two different redexes in a $\lambda^{\mathrm{MD}}$ term, they can only be disjoint, or one is a part of the other. In short, they are never overlapped each other. So, we can reduce one of them after we reduce another.

Every typed term is a value or can be reduced to another typed term.

**Theorem 8 (Progress).** *If $x : \tau@\epsilon \notin \Gamma$ and $\Gamma \vdash_{\Sigma} M : \tau@A$ then $M \in V^{A}$ or $M'$ exists such that $M \longrightarrow M'$.*

*Proof.* We can prove by straightforward induction on derivations. Difficult cases are T-App, T-◀, and T-Ins. We need Inversion Lemmas for them.

For every typed term, we can find just one redex to reduce by the evaluation context or it is a value. This theorem is important because it ensure that the evaluation context decides a redex to reduce deterministically. Specifically, this theorem guarantee that when you write a interpreter using the evaluation context of $\lambda^{\mathrm{MD}}$, your interpreter works just as intended.

**Theorem 9 (Unique Decomposition).** *If $x : \tau@\epsilon \notin \Gamma$ and $\Gamma \vdash_{\Sigma} M : \tau@A$ then 1 or 2 is true.*

1. *$M \in V^{A}$*
2. *There exist an unique tuple of $B, E^{A}_{B}, R^{B}$ such that $M = E^{A}_{B}[R^{B}]$ ($B = \epsilon$ or $B = \beta$) .*

*Proof.* We can prove by straightforward induction on derivations. Difficult cases are T-App, T-◀, and T-Ins. We need Inversion Lemmas for them.

## 5 Related Work

- On Cross-Stage Persistence in Multi-Stage[4]
  CSP も入りました
- Eliminating Array Bound Checking Through Dependent Types[19]

- MetaML and Multi-stage Programming with Explicit Annotations[15]
- Idris, a general-purpose dependently typed programming language: Design and implementation[1]
- A Logical Foundation for Environment Classifiers[16]
- Environment classifiers[14]
- A framework for defining logics[5]
  $\Sigma$ の使い方を確認した
- The Design and Implementation of BER MetaOCaml - System Description[7]
- Refined Environment Classifiers[8]
- Staging with control: type-safe multi-stage programming with control operators[12]
- Partial evaluation and automatic program generation [6]
- Efficient multi-level generating extensions for program specialization [3]
- Dependent types in practical programming[20]
  Section of Application を読んだ。Dead Code Elimination や Loop Unrolling にも使えるらしい。
- A dependently typed assembly language[18]
  DTAL の定義と制約 solver を用いた型検査の定義
- Run-time code generation and Modal-ML [17]
- C and tcc: a language and compiler for dynamic code generation [13]
- Run-time bytecode specialization [11]
- A tour of Tempo: A program specializer for the C language [2]
- Optimizing ML with run-time code generation [9]
- Efficient incremental run-time specialization for free [10]

## 6 Conclusions

## References

1. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23**(5), 552–593 (2013)
2. Consel, C., Lawall, J.L., Le Meur, A.F.: A tour of tempo: A program specializer for the c language. Science of Computer Programming **52**(1-3), 341–370 (2004)
3. Glück, R., Jørgensen, J.: Efficient multi-level generating extensions for program specialization. In: International Symposium on Programming Language Implementation and Logic Programming. pp. 259–278. Springer (1995)
4. Hanada, Y., Igarashi, A.: On Cross-Stage Persistence in Multi-Stage pp. 103–118 (2014)
5. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM (JACM) **40**(1), 143–184 (1993)
6. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Peter Sestoft (1993)
7. Kiselyov, O.: The design and implementation of BER metaocaml - system description. In: Codish, M., Sumii, E. (eds.) Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8475, pp. 86–102. Springer (2014). https://doi.org/10.1007/978-3-319-07151-0_6, https://doi.org/10.1007/978-3-319-07151-0_6

8. Kiselyov, O., Kameyama, Y., Sudo, Y.: Refined environment classifiers. In: Asian Symposium on Programming Languages and Systems. pp. 271–291. Springer (2016)
9. Lee, P., Leone, M.: Optimizing ML with run-time code generation, vol. 31. ACM (1996)
10. Marlet, R., Consel, C., Boinot, P.: Efficient incremental run-time specialization for free. In: ACM SIGPLAN Notices. vol. 34, pp. 281–292. ACM (1999)
11. Masuhara, H., Yonezawa, A.: Run-time bytecode specialization. In: Symposium on Program as Data Objects. pp. 138–154. Springer (2001)
12. Oishi, J., Kameyama, Y.: Staging with control: type-safe multi-stage programming with control operators. In: ACM SIGPLAN Notices. vol. 52, pp. 29–40. ACM (2017)
13. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, M.F.: C and tcc: a language and compiler for dynamic code generation. ACM Transactions on Programming Languages and Systems (TOPLAS) **21**(2), 324–369 (1999)
14. Taha, W., Nielsen, M.F.: Environment classifiers. In: ACM SIGPLAN Notices. vol. 38, pp. 26–37. ACM (2003)
15. Taha, W., Sheard, T.: Metaml and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1-2), 211–242 (Oct 2000). https://doi.org/10.1016/S0304-3975(00)00053-0, http://dx.doi.org/10.1016/S0304-3975(00)00053-0
16. Tsukada, T., Igarashi, A.: A Logical Foundation for Environment Classifiers. Logical Methods in Computer Science **Volume 6, Issue 4** (Dec 2010). https://doi.org/10.2168/LMCS-6(4:8)2010, https://lmcs.episciences.org/1065
17. Wickline, P., Lee, P., Pfenning, F.: Run-time code generation and Modal-ML, vol. 33. ACM (1998)
18. Xi, H., Harper, R.: A dependently typed assembly language. In: ACM SIGPLAN Notices. vol. 36, pp. 169–180. ACM (2001)
19. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. pp. 249–257. PLDI '98, ACM, New York, NY, USA (1998). https://doi.org/10.1145/277650.277732, http://doi.acm.org/10.1145/277650.277732
20. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 214–227. ACM (1999)

We solicit submissions in the form of regular research papers describing original scientific research results, including system development and case studies. Regular research papers should not exceed 18 pages in the Springer LNCS format, including bibliography and figures. This category encompasses both theoretical and implementation (also known as system descriptions) papers. In either case, submissions should clearly identify what has been accomplished and why it is significant. Submissions will be judged on the basis of significance, relevance, correctness, originality, and clarity. System descriptions papers should contain a link to a working system and will be judged on originality, usefulness, and design. In case of lack of space, proofs, experimental results, or any information supporting the technical results of the paper could be provided as an appendix or a link to a web page, but reviewers are not obliged to read them.