A dependently typed multi-stage calculus

AUTHOR(S):

Kawata, Akira; Igarashi, Atsushi

CITATION:

Kawata, Akira ...[et al]. A dependently typed multi-stage calculus. Programming Languages and Systems 2019, 11893: 53-72

ISSUE DATE:

2019-01-01

URL:

http://hdl.handle.net/2433/245912

A Dependently Typed Multi-Stage Calculus

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

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

Abstract. We study a dependently typed extension of a multi-stage programming language à la MetaOCaml, which supports quasi-quotation and cross-stage persistence for manipulation of code fragments as first-class values and an evaluation construct for execution of programs dynamically generated by this code manipulation. Dependent types are expected to bring to multi-stage programming enforcement of strong invariant—beyond simple type safety—on the behavior of dynamically generated code. An extension is, however, not trivial because such a type system would have to take stages of types—roughly speaking, the number of surrounding quotations—into account.
To rigorously study properties of such an extension, we develop λ^{MD}, which is an extension of Hanada and Igarashi's typed calculus $\lambda^{\triangleright\%}$ with dependent types, and prove its properties including preservation, confluence, strong normalization for full reduction, and progress for staged reduction. Motivated by code generators that generate code whose type depends on a value from outside of the quotations, we argue the significance of cross-stage persistence in dependently typed multi-stage programming and certain type equivalences that are not directly derived from reduction rules.

Keywords: multi-stage programming, cross-stage persistence, dependent types

1 Introduction

1.1 Multi-stage Programming and MetaOCaml

Multi-stage programming makes it easier for programmers to implement generation and execution of code at run time by providing language constructs for composing and running pieces of code as first-class values. A promising application of multi-stage programming is (run-time) code specialization, which generates program code specialized to partial inputs to the program and such applications are studied in the literature [17, 20, 29].

MetaOCaml [6, 18] is an extension of OCaml[1] with special constructs for multi-stage programming, including brackets and escape, which are (hygienic)

[1] http://ocaml.org

quasi-quotation, and `run`, which is similar to `eval` in Lisp, and cross-stage persistence (CSP) [31]. Programmers can easily write code generators by using these features. Moreover, MetaOCaml is equipped with a powerful type system for safe code generation and execution. The notion of code types is introduced to prevent code values that represent ill-typed expressions from being generated. For example, a quotation of expression `1 + 1` is given type `int code` and a code-generating function, which takes a code value c as an argument and returns c `+` c, is given type `int code -> int code` so that it cannot be applied to, say, a quotation of `"Hello"`, which is given type `string code`. Ensuring safety for `run` is more challenging because code types by themselves do not guarantee that the execution of code values never results in unbound variable errors. Taha and Nielsen [30] introduced the notion of environment classifiers to address the problem, developed a type system to ensure not only type-safe composition but also type-safe execution of code values, and proved a type soundness theorem (for a formal calculus λ^{α} modeling a pure subset of MetaOCaml).

However, the type system, which is based on the Hindley–Milner polymorphism [23], is not strong enough to guarantee invariant beyond simple types. For example, Kiselyov [17] demonstrates specialization of vector/matrix computation with respect to the sizes of vectors and matrices in MetaOCaml but the type system of MetaOCaml cannot prevent such specialized functions from being applied to vectors and matrices of different sizes.

1.2 Multi-stage Programming with Dependent Types

One natural idea to address this problem is the introduction of dependent types to express the size of data structures in static types [34]. For example, we could declare vector types indexed by the size of vectors as follows.

```
Vector :: Int -> *
```

`Vector` is a type constructor that takes an integer (which represents the length of vectors): for example, `Vector 3` is the type for vectors whose lengths are 3. Then, our hope is to specialize vector/matrix functions with respect to their size and get a piece of function code, whose type respects the given size, *provided at specialization time*. For example, we would like to specialize a function to add two vectors with respect to the size of vectors, that is, to implement a code generator that takes a (nonnegative) integer n as an input and generates a piece of function code of type `(Vector` n `-> Vector` n `-> Vector` n`) code`.

1.3 Our Work

In this paper, we develop a new multi-stage calculus λ^{MD} by extending the existing multi-stage calculus $\lambda^{\triangleright\%}$ [14] with dependent types and study its properties. We base our work on $\lambda^{\triangleright\%}$, in which the four multi-stage constructs are handled slightly differently from MetaOCaml, because its type system and semantics are arguably simpler than λ^{α} [30], which formalizes the design of MetaOCaml more

faithfully. Dependent types are based on λLF [1], which has one of the simplest forms of dependent types. Our technical contributions are summarized as follows:

- We give a formal definition of λ^{MD} with its syntax, type system and two kinds of reduction: full reduction, allowing reduction of any redex, including one under λ-abstraction and quotation, and staged reduction, a small-step call-by-value operational semantics that is closer to the intended multi-stage implementation.
- We show preservation, strong normalization, and confluence for full reduction; and show unique decomposition (and progress as its corollary) for staged reduction.

The combination of multi-stage programming and dependent types has been discussed by Pasalic, Taha, and Sheard [26] and Brady and Hammond [5] but, to our knowledge, our work is a first formal calculus of full-spectrum dependently typed multi-stage programming with all the key constructs mentioned above.

Organization of the Paper. The organization of this paper is as follows. Section 2 gives an informal overview of λ^{MD}. Section 3 defines λ^{MD} and Section 4 shows properties of λ^{MD}. Section 5 discusses related work and Section 6 concludes the paper with discussion of future work. We omit proofs and (details of) some definitions for brevity; interested readers are referred to a full version of the paper, which is available at https://arxiv.org/abs/1908.02035.

2 Informal Overview of λ^{MD}

We describe our calculus λ^{MD} informally. λ^{MD} is based on $\lambda^{\triangleright\%}$ [14] by Hanada and Igarashi and so we start with a review of $\lambda^{\triangleright\%}$.

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

In $\lambda^{\triangleright\%}$, brackets (quasi-quotation) and escape (unquote) are written $\blacktriangleright_\alpha M$ and $\blacktriangleleft_\alpha M$, respectively. For example, $\blacktriangleright_\alpha(1+1)$ represents code of expression $1+1$ and thus evaluates to itself. Escape $\blacktriangleleft_\alpha M$ may appear under $\blacktriangleright_\alpha$; it evaluates M to a code value and splices it into the surrounding code fragment. Such splicing is expressed by the following reduction rule:

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

The subscript α in $\blacktriangleright_\alpha$ and $\blacktriangleleft_\alpha$ is a *stage variable*[2] and a sequence of stage variables is called a *stage*. Intuitively, a stage represents the depth of nested brackets. Stage variables can be abstracted by $\Lambda\alpha.M$ and instantiated by an

[2] In Hanada and Igarashi [14], it was called a *transition variable*, which is derived from correspondence to modal logic, studied by Tsukada and Igarashi [32].

application $M\ A$ to stages. For example, $\Lambda\alpha.\blacktriangleright_\alpha((\lambda x : \mathrm{Int}.x + 10)\ 5)$ is a code value, where α is abstracted. If it is applied to $A = \alpha_1 \cdots \alpha_n$, $\blacktriangleright_\alpha$ becomes $\blacktriangleright_{\alpha_1} \cdots \blacktriangleright_{\alpha_n}$; in particular, if $n = 0$, $\blacktriangleright_\alpha$ disappears. So, an application of $\Lambda\alpha.\blacktriangleright_\alpha((\lambda x : \mathrm{Int}.x + 10)\ 5)$ to the empty sequence ε reduces to (unquoted) $(\lambda x : \mathrm{Int}.x + 10)\ 5$ and to 15. In other words, application of a Λ-abstraction to ε corresponds to `run`. This is expressed by the following reduction rule:

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

where stage substitution $[\alpha \mapsto A]$ manipulates the nesting of $\blacktriangleright_\alpha$ and $\blacktriangleleft_\alpha$ (and also $\%_\alpha$ as we see later).

Cross-stage persistence (CSP), which is an important feature of $\lambda^{\triangleright\%}$, is a primitive to embed values (not necessarily code values) into a code value. For example, a $\lambda^{\triangleright\%}$-term

$$M_1 = \lambda x : \mathrm{Int}.\Lambda\alpha.(\blacktriangleright_\alpha((\%_\alpha x) * 2))$$

takes an integer x as an input and returns a code value, into which x is embedded. If M_1 is applied to $38 + 4$ as in

$$M_2 = (\lambda x : \mathrm{Int}.\Lambda\alpha.(\blacktriangleright_\alpha((\%_\alpha x) * 2)))\ (38 + 4),$$

then it evaluates to $M_3 = \Lambda\alpha.(\blacktriangleright_\alpha((\%_\alpha 42) * 2))$. According to the semantics of $\lambda^{\triangleright\%}$, the subterm $\%_\alpha 42$ means that it waits for the surrounding code to be run (by an application to ε) and so it does not reduce further. If M_3 is run by application to ε, substitution of ε for α eliminates $\blacktriangleright_\alpha$ and $\%_\alpha$ and so $42 * 2$, which reduces to 84, is obtained. CSP is practically important because one can call library functions from inside quotations.

The type system of $\lambda^{\triangleright\%}$ uses code types—the type of code of type τ is written $\triangleright_\alpha\tau$—for typing $\blacktriangleright_\alpha$, $\blacktriangleleft_\alpha$ and $\%_\alpha$. It takes stages into account: a variable declaration (written $x : \tau@A$) in a type environment is associated with its declared stage A as well as its type τ and the type judgement of $\lambda^{\triangleright\%}$ is of the form $\Gamma \vdash M : \tau@A$, in which A stands for the stage of term M.[3] For example, $y : \mathrm{Int}@\alpha \vdash (\lambda x : \mathrm{Int}.y) : \mathrm{Int} \to \mathrm{Int}@\alpha$ holds, but $y : \mathrm{Int}@\alpha \vdash (\lambda x : \mathrm{Int}.y) : \mathrm{Int} \to \mathrm{Int}@\varepsilon$ does not because the latter uses y at stage ε but y is declared at α. Quotation $\blacktriangleright_\alpha M$ is given type $\triangleright_\alpha\tau$ at stage A if M is given type τ at stage $A\alpha$; unquote $\blacktriangleleft_\alpha M$ is given type τ at stage $A\alpha$ if M is given type $\triangleright_\alpha\tau$ at stage $A\alpha$; and CSP $\%_\alpha M$ is give type τ at stage $A\alpha$ if M is given type τ at A. These are expressed by the following typing rules.

$$\frac{\Gamma \vdash M : \tau@A\alpha}{\Gamma \vdash \blacktriangleright_\alpha M : \triangleright_\alpha\tau@A} \qquad \frac{\Gamma \vdash M : \triangleright_\alpha\tau@A}{\Gamma \vdash \blacktriangleleft_\alpha M : \tau@A\alpha} \qquad \frac{\Gamma \vdash M : \tau@A}{\Gamma \vdash \%_\alpha M : \tau@A\alpha}$$

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

In this paper, we add a simple form of dependent types—à la Edinburgh LF [15] and λLF [1]—to $\lambda^{\triangleright\%}$. Types can be indexed by terms as in `Vector` in Section 1

[3] In Hanada and Igarashi [14], it is written $\Gamma \vdash^A M : \tau$.

and λ-abstractions can be given dependent function types of the form $\Pi x : \tau.\sigma$ but we do not consider type operators (such as `list` τ) or abstraction over type variables. We introduce kinds to classify well-formed types and equivalences for kinds, types, and terms—as in other dependent type systems—but we have to address a question how the notion of stage (should) interact with kinds and types.

On the one hand, base types such as Int should be able to be used at every stage as in $\lambda^{\triangleright\%}$ so that $\lambda x : \mathrm{Int}.\Lambda\alpha.\blacktriangleright_\alpha \lambda y : \mathrm{Int}.M$ is a valid term (here, Int is used at ε and α). Similarly for indexed types such as Vector 4. On the other hand, it is not immediately clear how a type indexed by a variable, which can be used only at a declared stage, can be used. For example, consider

$$\blacktriangleright_\alpha(\lambda x : \mathrm{Int}.(\blacktriangleleft_\alpha(\lambda y : \mathrm{Vector}\ x.M)N)) \text{ and } \lambda x : \mathrm{Int}.\blacktriangleright_\alpha(\lambda y : \mathrm{Vector}\ x.M).$$

Is Vector x a legitimate type at ε (and α, resp.) even if x : Int is declared at stage α (and ε, resp.)? We will give our answer to this question in two steps.

First, type-level constants such as Int and Vector can be used at every stage in λ^{MD}. Technically, we introduce a signature that declares kinds of type-level constants and types of constants. For example, a signature for the Boolean type and constants is given as follows $\mathrm{Bool} :: *, \mathrm{true} : \mathrm{Bool}, \mathrm{false} : \mathrm{Bool}$ (where $*$ is the kind of proper types). Declarations in a signature are not associated to particular stages; so they can be used at every stage.

Second, an indexed type such as Vector 3 or Vector x is well formed only at the stage(s) where the index term is well-typed. Since constant 3 is well-typed at every stage (if it is declared in the signature), Vector 3 is a well-formed type at every stage, too. However, Vector M is well-formed only at the stage where index term M is typed. Thus, the kinding judgment of λ^{MD} takes the form $\Gamma \vdash_\Sigma \tau :: K@A$, where stage A stands for where τ is well-formed. For example, given Vector :: Int $\to *$ in the signature Σ, $x : \mathrm{Int}@\varepsilon \vdash_\Sigma \mathrm{Vector}\ x :: *@\varepsilon$ can be derived but neither $x : \mathrm{Int}@\alpha \vdash_\Sigma \mathrm{Vector}\ x :: *@\varepsilon$ nor $x : \mathrm{Int}@\varepsilon \vdash_\Sigma \mathrm{Vector}\ x :: *@\alpha$ can be.

Apparently, the restriction above sounds too severe, because a term like $\lambda x : \mathrm{Int}.\blacktriangleright_\alpha(\lambda y : \mathrm{Vector}\ x.M)$, which models a typical code generator which takes the size x and returns code for vector manipulation specialized to the given size, will be rejected. It seems crucial for y to be given a type indexed by x. We can address this problem by CSP—In fact, Vector x is not well formed at α under $x : \mathrm{Int}@\varepsilon$ but Vector $(\%_\alpha x)$ is! Thus, we can still write $\lambda x : \mathrm{Int}.\blacktriangleright_\alpha(\lambda y : \mathrm{Vector}\ (\%_\alpha x).M)$ for the typical sort of code generators.

Our decision that well-formedness of types takes stages of index terms into account will lead to the introduction of CSP at the type level and special equivalence rules, as we will see later.

3 Formal Definition of λ^{MD}

In this section, we give a formal definition of λ^{MD}, including the syntax, full reduction, and type system. In addition to the full reduction, in which any redex

at any stage can be reduced, we also give staged reduction, which models program execution (at ε-stage).

3.1 Syntax

We assume the denumerable set of *type-level constants*, ranged over by metavariables X, Y, Z, the denumerable set of *variables*, ranged over by x, y, z, the denumerable set of *constants*, ranged over by c, and the denumerable set of *stage variables*, ranged over by α, β, γ. The metavariables A, B, C range over sequences of stage variables; we write ε for the empty sequence. λ^{MD} is defined by the following grammar:

$$
\begin{array}{rr}
\text{kinds} & K, J, I, H, G ::= * \mid \Pi x : \tau.K \\
\text{types} & \tau, \sigma, \rho, \pi, \xi ::= X \mid \Pi x : \tau.\sigma \mid \tau\ M \mid \triangleright_\alpha \tau \mid \forall \alpha.\tau \\
\text{terms} & M, N, L, O, P ::= c \mid x \mid \lambda x : \tau.M \mid M\ N \mid \blacktriangleright_\alpha M \\
 & \qquad \mid \blacktriangleleft_\alpha M \mid \Lambda \alpha.M \mid M\ A \mid \%_\alpha M \\
\text{signatures} & \Sigma ::= \emptyset \mid \Sigma, X :: K \mid \Sigma, c : \tau \\
\text{type env.} & \Gamma ::= \emptyset \mid \Gamma, x : \tau @ A
\end{array}
$$

A kind, which is used to classify types, is either $*$, the kind of proper types (types that terms inhabit), or $\Pi x\colon \tau.K$, the kind of type operators that takes x as an argument of type τ and returns a type of kind K. A type is a type-level constant X, which is declared in the signature with its kind, a dependent function type $\Pi x : \tau.\sigma$, an application $\tau\ M$ of a type (operator of Π-kind) to a term, a code type $\triangleright_\alpha \tau$, or an α-closed type $\forall \alpha.\tau$. An example of an application of a type (operator) of Π-kind to a term is Vector 10; it is well kinded if, say, the type-level constant Vector has kind $\Pi x : \mathrm{Int}.*$. A code type $\triangleright_\alpha \tau$ is for a code fragment of a term of type τ. An α-closed type, when used with $\triangleright_\alpha$, represents runnable code.

Terms include ordinary (explicitly typed) λ-terms, constants, whose types are declared in signature Σ, and the following five forms related to multi-stage programming: $\blacktriangleright_\alpha M$ represents a code fragment; $\blacktriangleleft_\alpha M$ represents escape; $\Lambda \alpha.M$ is a stage variable abstraction; $M\ A$ is an application of a stage abstraction M to stage A; and $\%_\alpha M$ is an operator for cross-stage persistence.

We adopt the tradition of λLF-like systems, where types of constants and kinds of type-level constants are globally declared in a signature Σ, which is a sequence of declarations of the form $c : \tau$ and $X :: K$. For example, when we use Boolean in λ^{MD}, Σ includes $\mathrm{Bool} :: *, \mathrm{true} : \mathrm{Bool}, \mathrm{false} : \mathrm{Bool}$. Type environments are sequences of triples of a variable, its type, and its stage. We write $\mathit{dom}(\Sigma)$ and $\mathit{dom}(\Gamma)$ for the set of (type-level) constants and variables declared in Σ and Γ, respectively. As in other multi-stage calculi [14, 30, 32], a variable declaration is associated with a stage so that a variable can be referenced

only at the declared stage. On the contrary, constants and type-level constants are *not* associated with stages; so, they can appear at any stage. We define well-formed signatures and well-formed type environments later.

The variable x is bound in M by $\lambda x : \tau.M$ and in σ by $\Pi x : \tau.\sigma$, as usual; the stage variable α is bound in M by $\Lambda\alpha.M$ and τ by $\forall\alpha.\tau$. The notion of free variables is defined in a standard manner. 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. Similarly, $\mathrm{FV}(\tau)$, $\mathrm{FSV}(\tau)$, $\mathrm{FV}(K)$, and $\mathrm{FSV}(K)$ are defined. We sometimes abbreviate $\Pi x : \tau_1.\tau_2$ to $\tau_1 \to \tau_2$ if x is not a free variable of τ_2. We identify α-convertible terms and assume the names of bound variables are pairwise distinct.

The prefix operators $\triangleright_\alpha, \blacktriangleright_\alpha, \blacktriangleleft_\alpha$, and $\%_\alpha$ are given higher precedence over the three forms $\tau\ M$, $M\ N$, $M\ A$ of applications, which are left-associative. The binders Π, $\forall$, and λ extend as far to the right as possible. Thus, $\forall\alpha.\triangleright_\alpha(\Pi x : \mathrm{Int}.\mathrm{Vector}\ 5)$ is interpreted as $\forall\alpha.(\triangleright_\alpha(\Pi x : \mathrm{Int}.(\mathrm{Vector}\ 5)))$; and $\Lambda\alpha.\lambda x : \mathrm{Int}.\blacktriangleright_\alpha x\ y$ means $\Lambda\alpha.(\lambda x : \mathrm{Int}.(\blacktriangleright_\alpha x)\ y)$.

Remark: Basically, we define λ^{MD} to be an extension of $\lambda^{\triangleright\%}$ with dependent types. One notable difference is that λ^{MD} has only one kind of α-closed types, whereas $\lambda^{\triangleright\%}$ has two kinds of α-closed types $\forall\alpha.\tau$ and $\forall^\varepsilon\alpha.\tau$. We have omitted the first kind, for simplicity, and dropped the superscript ε from the second. It would not be difficult to recover the distinction to show properties related to program residualization [14], although they are left as conjectures.

3.2 Reduction

Next, we define full reduction for λ^{MD}. Before giving the definition of reduction, we define two kinds of substitutions. Substitution $M[x \mapsto N], \tau[x \mapsto N]$ and $K[x \mapsto N]$ are ordinary capture-avoiding substitution of term N for x in term M, type τ, and kind K, respectively, and we omit their definitions here. Substitution $M[\alpha \mapsto A], \tau[\alpha \mapsto A], K[\alpha \mapsto A]$ and $B[\alpha \mapsto A]$ are substitutions of stage A for stage variable α in term M, type τ, kind K, and stage B, respectively. We show representative cases below.

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

Here, $\blacktriangleright_{\alpha_1 \cdots \alpha_n} M$, $\blacktriangleleft_{\alpha_1 \cdots \alpha_n} M$, and $\%_{\alpha_1 \cdots \alpha_n} M$ $(n \geq 0)$ stand for $\blacktriangleright_{\alpha_1} \cdots \blacktriangleright_{\alpha_n} M$, $\blacktriangleleft_{\alpha_n} \cdots \blacktriangleleft_{\alpha_1} M$, and $\%_{\alpha_n} \cdots \%_{\alpha_1} M$, respectively. In particular, $\blacktriangleright_\varepsilon M = \blacktriangleleft_\varepsilon M = \%_\varepsilon M = M$. Also, it is important that the order of stage variables is reversed for $\blacktriangleleft$ and $\%$. We also define substitutions of a stage or a term for variables in type environment Γ.

Definition 1 (Reduction). *The relations* $M \longrightarrow_\beta N$, $M \longrightarrow_\blacklozenge N$, *and* $M \longrightarrow_\Lambda N$ *are the least compatible relations closed under the rules below.*

$$(\lambda x : \tau.M)N \longrightarrow_\beta M[x \mapsto N]$$
$$\blacktriangleleft_\alpha \blacktriangleright_\alpha M \longrightarrow_\blacklozenge M$$
$$(\Lambda\alpha.M)\ A \longrightarrow_\Lambda M[\alpha \mapsto A]$$

We write $M \longrightarrow M'$ iff $M \longrightarrow_\beta M'$, $M \longrightarrow_\blacklozenge M'$, or $M \longrightarrow_\Lambda M'$ and we call $\longrightarrow_\beta$, $\longrightarrow_\blacklozenge$, and $\longrightarrow_\Lambda$ β-reduction, $\blacklozenge$-reduction, and Λ-reduction, respectively. $M \longrightarrow^* N$ means that there is a sequence of reduction $\longrightarrow$ whose length is greater than or equal to 0.

The relation $\longrightarrow_\beta$ represents ordinary β-reduction in the λ-calculus; the relation $\longrightarrow_\blacklozenge$ represents that quotation $\blacktriangleright_\alpha M$ is canceled by escape and M is spliced into the code fragment surrounding the escape; the relation $\longrightarrow_\Lambda$ means that a stage abstraction applied to stage A reduces to the body of the abstraction where A is substituted for the stage variable. There is no reduction rule for CSP as with Hanada and Igarashi [14]. The CSP operator $\%_\alpha$ disappears when ε is substituted for α. We show an example of a reduction sequence below. Underlines show the redexes.

$$\begin{aligned}
&\underline{(\lambda f : \mathrm{Int} \to \mathrm{Int}.(\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha f\ 1 + (\blacktriangleleft_\alpha \blacktriangleright_\alpha 3))\ \varepsilon))\ (\lambda x : \mathrm{Int}.x)} \\
\longrightarrow_\beta\ &(\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha(\lambda x : \mathrm{Int}.x)\ 1 + (\underline{\blacktriangleleft_\alpha \blacktriangleright_\alpha 3}))\ \varepsilon \\
\longrightarrow_\blacklozenge\ &\underline{(\Lambda\alpha.\blacktriangleright_\alpha(\%_\alpha(\lambda x : \mathrm{Int}.x)\ 1 + 3))\ \varepsilon} \\
\longrightarrow_\Lambda\ &\underline{(\lambda x : \mathrm{Int}.x)\ 1} + 3 \\
\longrightarrow_\beta\ &1 + 3 \\
\longrightarrow^*\ &4
\end{aligned}$$

3.3 Type System

In this section, we define the type system of λ^{MD}. It consists of eight judgment forms for signature well-formedness, type environment well-formedness, kind well-formedness, kinding, typing, kind equivalence, type equivalence, and term equivalence. We list the judgment forms in Figure 1. They are all defined in a mutual recursive manner. We will discuss each judgment below.

Signature and Type Environment Well-formedness. The rules for Well-formed signatures and type environments are shown below:

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Sigma \qquad \vdash_\Sigma K\ \mathrm{kind}@\varepsilon \qquad X \notin dom(\Sigma)}{\vdash \Sigma, X :: K} \qquad \frac{\vdash \Sigma \qquad \vdash_\Sigma \tau :: *@\varepsilon \qquad c \notin dom(\Sigma)}{\vdash \Sigma, c : \tau}$$

$$\frac{}{\vdash_\Sigma \emptyset} \qquad \frac{\vdash_\Sigma \Gamma \qquad \Gamma \vdash_\Sigma \tau :: *@A \qquad x \notin dom(\Sigma)}{\vdash_\Sigma \Gamma, x : \tau@A}$$

$\vdash \Sigma$	signature well-formedness
$\vdash_\Sigma \Gamma$	type environment well-formedness
$\Gamma \vdash_\Sigma K$ kind@A	kind well-formedness
$\Gamma \vdash_\Sigma \tau :: K@A$	kinding
$\Gamma \vdash_\Sigma M : \tau@A$	typing
$\Gamma \vdash_\Sigma K \equiv J@A$	kind equivalence
$\Gamma \vdash_\Sigma \tau \equiv \sigma :: K@A$	type equivalence
$\Gamma \vdash_\Sigma M \equiv N : \tau@A$	term equivalence

Fig. 1. Eight judgment forms of the type system of λ^{MD}.

To add declarations to a signature, the kind/type of a (type-level) constant has to be well-formed at stage ε so that it is used at any stage. In what follows, well-formedness is not explicitly mentioned but we assume that all signatures and type environments are well-formed.

Kind Well-formedness and Kinding. The rules for kind well-formedness and kinding are a straightforward adaptation from λLF and $\lambda^{\triangleright\%}$, except for the following rule for type-level CSP.

$$\frac{\Gamma \vdash_\Sigma \tau :: *@A}{\Gamma \vdash_\Sigma \tau :: *@A\alpha}\ (\text{K-Csp})$$

Unlike the term level, type-level CSP is implicit because there is no staged semantics for types.

Typing. The typing rules of λ^{MD} are shown in Figure 2. The rule T-Const means that a constant can appear at any stage. The rules T-Var, T-Abs, and T-App are almost the same as those in the simply typed lambda calculus or λLF. Additional conditions are that subterms must be typed at the same stage (T-Abs and T-App); the type annotation/declaration on a variable has to be a proper type of kind $*$ (T-Abs) at the stage where it is declared (T-Var and T-Abs).

As in standard dependent type systems, T-Conv allows us to replace the type of a term with an equivalent one. For example, assuming integers and arithmetic, a value of type Vector $(4 + 1)$ can also have type Vector 5 because of T-Conv.

The rules T-$\blacktriangleright$, T-$\blacktriangleleft$, T-Gen, T-Ins, and T-Csp are constructs for multi-stage programming. T-$\blacktriangleright$ and T-$\blacktriangleleft$ are the same as in $\lambda^{\triangleright\%}$, as we explained in Section 2. The rule T-Gen for stage abstraction is straightforward. The condition $\alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\mathrm{A})$ ensures that the scope of α is in M, and avoids

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

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

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

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

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

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

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

Fig. 2. Typing Rules.

capturing variables elsewhere. The rule T-Ins is for applications of stages to stage abstractions. The rule T-Csp is for CSP, which means that, if term M is of type τ at stage A, then $\%_\alpha M$ is of type τ at stage $A\alpha$. Note that CSP is also applied to the type τ (although it is implicit) in the conclusion. Thanks to implicit CSP, the typing rule is the same as in $\lambda^{\triangleright\%}$.

Kind, Type and Term Equivalence. Since the syntax of kinds, types, and terms is mutually recursive, the corresponding notions of equivalence are also mutually recursive. They are congruences closed under a few axioms for term equivalence. Thus, the rules for kind and type equivalences are not very interesting, except that implicit CSP is allowed. We show a few representative rules below.

$$\frac{\Gamma \vdash_\Sigma K \equiv J@A}{\Gamma \vdash_\Sigma K \equiv J@A\alpha}\ (\text{QK-Csp}) \qquad \frac{\Gamma \vdash_\Sigma \tau \equiv \sigma :: *@A}{\Gamma \vdash_\Sigma \tau \equiv \sigma :: *@A\alpha}\ (\text{QT-Csp})$$

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

We show the rules for term equivalence in Figure 3, omitting straightforward rules for reflexivity, symmetry, transitivity, and compatibility. The rules Q-β, Q-$\blacktriangleleft\blacktriangleright$, and Q-$\Lambda$ correspond to β-reduction, $\blacklozenge$-reduction, and Λ-reduction, respectively.

The only rule that deserves elaboration is the last rule Q-%. Intuitively, it means that the CSP operator applied to term M can be removed if M is also

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

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

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

Fig. 3. Term Equivalence Rules.

well-typed at the next stage $A\alpha$. For example, constants do not depend on the stage (see T-Const) and so $\Gamma \vdash_\Sigma \%_\alpha c \equiv c : \tau@A\alpha$ holds but variables do depend on stages and so this rule does not apply.

Example. We show an example of a dependently typed code generator in a hypothetical language based on $\lambda^{\triangleright\%}$. This language provides definitions by **let**, recursive functions (represented by **fix**), **if**-expressions, and primitives cons, head, and tail to manipulate vectors. We assume that cons is of type $\Pi n : \text{Int}.\text{Int} \to \text{Vector}\ n \to \text{Vector}\ (n+1)$, head is of type $\Pi n : \text{Int}.\text{Vector}\ (n+1) \to \text{Int}$, and tail is of type $\Pi n : \text{Int}.\text{Vector}\ (n+1) \to (\text{Vector}\ n)$.

Let's consider an application, for example, in computer graphics, in which we have potentially many pairs of vectors of the fixed (but statically unknown) length and a function—such as vector addition—to be applied to them. This function should be fast because it is applied many times and be safe because just one runtime error may ruin the whole long-running calculation.

Our goal is to define the function vadd of type

$$\Pi n : \text{Int}.\forall\beta.\triangleright_\beta(\text{Vector}\ (\%_\alpha n) \to \text{Vector}\ (\%_\alpha n) \to \text{Vector}\ (\%_\alpha n)).$$

It takes the length n and returns (β-closed) code of a function to add two vectors of length n. The generated code is run by applying it to ε to obtain a function of type $\text{Vector}\ n \to \text{Vector}\ n \to \text{Vector}\ n$ as expected.

We start with the helper function vadd_1, which takes a stage, the length n of vectors, and two quoted vectors as arguments and returns code that computes the addition of the given two vectors:

$$\begin{array}{l}
\textbf{let}\ \text{vadd}_1 : \forall\alpha.\Pi n : \text{Int}.\triangleright_\alpha\text{Vector}\ n \to \triangleright_\alpha\text{Vector}\ n \to \triangleright_\alpha\text{Vector}\ n \\
\quad = \textbf{fix}\ f.\Lambda\alpha.\lambda n : \text{Int}.\ \lambda v_1 : \triangleright_\alpha\text{Vector}\ n.\ \lambda v_2 : \triangleright_\alpha\text{Vector}\ n. \\
\qquad \textbf{if}\ n = 0\ \textbf{then}\ \blacktriangleright_\alpha\text{nil} \\
\qquad \textbf{else}\ \blacktriangleright_\alpha(\ \textbf{let}\ t_1 = \text{tail}\ (\blacktriangleleft_\alpha v_1)\ \textbf{in} \\
\qquad\qquad\qquad \textbf{let}\ t_2 = \text{tail}\ (\blacktriangleleft_\alpha v_2)\ \textbf{in} \\
\qquad\qquad\qquad \text{cons}\ (\text{head}\ (\blacktriangleleft_\alpha v_1) + \text{head}\ (\blacktriangleleft_\alpha v_2)) \\
\qquad\qquad\qquad\qquad \blacktriangleleft_\alpha(f\ (n-1)\ (\blacktriangleright_\alpha t_1)\ (\blacktriangleright_\alpha t_2)))
\end{array}$$

Note that the generated code will not contain branching on n or recursion. (Here, we assume that the type system can determine whether $n = 0$ when

then- and **else**-branches are typechecked so that both branches can be given type $\triangleright_\alpha$Vector n.)

Using vadd_1, the main function vadd can be defined as follows:

$$\begin{aligned}&\textbf{let}\ \text{vadd} : \mathit{\Pi} n : \text{Int}.\forall\beta.\triangleright_\beta(\text{Vector}\ (\%_\beta n) \to \text{Vector}\ (\%_\beta n) \to \text{Vector}\ (\%_\beta n))\\ &\quad = \lambda n : \text{Int}.\mathit{\Lambda}\beta.\blacktriangleright_\beta(\lambda v_1 : \text{Vector}\ (\%_\beta n).\ \lambda v_2 : \text{Vector}\ (\%_\beta n).\\ &\qquad\qquad \blacktriangleleft_\beta(\text{vadd}_1\ \beta\ n\ (\blacktriangleright_\beta\ v_1)\ (\blacktriangleright_\beta\ v_2)))\end{aligned}$$

The auxiliary function vadd_1 generates code to compute addition of the formal arguments v_1 and v_2 without branching on n or recursion. As we mentioned already, if this function is applied to a (nonnegative) integer constant, say 5, it returns function code for adding two vectors of size 5. The type of vadd 5, obtained by substituting 5 for n, is $\forall\beta.\triangleright_\beta(\text{Vector}\ (\%_\beta 5) \to \text{Vector}\ (\%_\beta 5) \to \text{Vector}\ (\%_\beta 5))$. If the obtained code is run by applying to ε, the type of vadd 5 ε is Vector 5 $\to$ Vector 5 $\to$ Vector 5 as expected.

There are other ways to implement the vector addition function: by using tuples instead of lists if the length for all the vectors is statically known or by checking dynamically the lengths of lists for every pair. However, our method is better than these alternatives in two points. First, our function, vadd_1 can generate functions for vectors of arbitrary length unlike the one using tuples. Second, vadd_1 has an advantage in speed over the one using dynamic checking because it can generate an optimized function for a given length.

We make two technical remarks before proceeding:

1. If the generated function code is composed with another piece of code of type, say, $\triangleright_\gamma$Vector 5, Q-% plays an essential role; that is, Vector 5 and Vector $(\%_\gamma 5)$, which would occur by applying the generated code to γ (instead of ε), are syntactically different types but Q-% enables to equate them. Interestingly, Hanada and Igarashi [14] rejected the idea of reduction that removes $\%_\alpha$ when they developed $\lambda^{\triangleright\%}$, as such reduction does not match the operational behavior of the CSP operator in implementations. However, as an equational system for multi-stage programs, the rule Q-% makes sense.
2. By using implicit type-level CSP, the type of vadd could have been written $\mathit{\Pi} n$: Int.$\forall\beta.\triangleright_\beta$(Vector $n \to$ Vector $n \to$ Vector n). In this type, Vector n is given kind at stage ε and type-level CSP implicitly lifts it to stage β. However, if a type-level constant takes two or more arguments from different stages, term-level CSP is necessary. A matrix type (indexed by the numbers of columns and rows) would be such an example.

3.4 Staged Semantics

The reduction given above is full reduction and any redexes—even under $\blacktriangleright_\alpha$—can be reduced in an arbitrary order. Following previous work [14], we introduce (small-step, call-by-value) staged semantics, where only β-reduction or $\mathit{\Lambda}$-reduction at stage ε or the outer-most $\blacklozenge$-reduction are allowed, modeling an implementation.

We start with the definition of values. Since terms under quotations are not executed, the grammar is indexed by stages.

Definition 2 (Values). *The family V^A of sets of values, ranged over by v^A, is defined by the following grammar. In the grammar, $A' \neq \varepsilon$ is assumed.*

$$\begin{aligned} v^\varepsilon \in V^\varepsilon ::= {} & \lambda x : \tau.M \mid \ \blacktriangleright_\alpha v^\alpha \mid \Lambda\alpha.v^\varepsilon \\ v^{A'} \in V^{A'} ::= {} & x \mid \lambda x : \tau.v^{A'} \mid v^{A'}\ v^{A'} \mid \ \blacktriangleright_\alpha v^{A'\alpha} \mid \Lambda\alpha.v^{A'} \mid v^{A'}\ B \\ & \mid \ \blacktriangleleft_\alpha v^{A''} (\textit{if } A' = A''\alpha \textit{ for some } \alpha, A'' \neq \varepsilon) \\ & \mid \ \%_\alpha v^{A''} (\textit{if } A' = A''\alpha) \end{aligned}$$

Values at stage ε are λ-abstractions, quoted pieces of code, or Λ-abstractions. The body of a λ-abstraction can be any term but the body of Λ-abstraction has to be a value. It means that the body of Λ-abstraction must be evaluated. The side condition for $\blacktriangleleft_\alpha v^{A'}$ means that escapes in a value can appear only under nested quotations because an escape under a single quotation will splice the code value into the surrounding code. See Hanada and Igarashi [14] for details.

In order to define staged reduction, we define redex and evaluation contexts.

Definition 3 (Redex). *The sets of ε-redexes (ranged over by R^ε) and α-redexes (ranged over by R^α) are defined by the following grammar.*

$$\begin{aligned} R^\varepsilon ::= {} & (\lambda x : \tau.M)\ v^\varepsilon \mid (\Lambda\alpha.v^\varepsilon)\ \varepsilon \\ R^\alpha ::= {} & \ \blacktriangleleft_\alpha \blacktriangleright_\alpha v^\alpha \end{aligned}$$

Definition 4 (Evaluation Context). *Let B be either ε or a stage variable β. The family of sets $ECtx_B^A$ of evaluation contexts, ranged over by E_B^A, is defined by the following grammar (in which A' stands for a non-empty stage).*

$$\begin{aligned} E_B^\varepsilon \in ECtx_B^\varepsilon ::= {} & \Box\ (\textit{if } B = \varepsilon) \mid E_B^\varepsilon\ M \mid v^\varepsilon\ E_B^\varepsilon \mid \blacktriangleright_\alpha E_B^\alpha \mid \Lambda\alpha.E_B^\varepsilon \mid E_B^\varepsilon\ A \\ E_B^{A'} \in ECtx_B^{A'} ::= {} & \Box\ (\textit{if } A' = B) \mid \lambda x : \tau.E_B^{A'} \mid E_B^{A'}\ M \mid v^{A'}\ E_B^{A'} \\ & \mid \blacktriangleright_\alpha E_B^{A'\alpha} \mid \blacktriangleleft_\alpha E_B^A\ (\textit{where } A\alpha = A') \\ & \mid \Lambda\alpha.E_B^{A'} \mid E_B^{A'}\ A \mid \%_\alpha\ E_B^A\ (\textit{where } A\alpha = A') \end{aligned}$$

The subscripts A and B in E_B^A stand for the stage of the evaluation context and of the hole, respectively. The grammar represents that staged reduction is left-to-right and call-by-value and terms under Λ are reduced. Terms at non-ε stages are not reduced, except redexes of the form $\blacktriangleleft_\alpha \blacktriangleright_\alpha v^\alpha$ at stage α. A few examples of evaluation contexts are shown below:

$$\begin{aligned} \Box\ (\lambda x : \mathrm{Int}.x) &\in ECtx_\varepsilon^\varepsilon \\ \Lambda\alpha.\Box\ \varepsilon &\in ECtx_\varepsilon^\varepsilon \\ \blacktriangleleft_\alpha \blacktriangleright_\alpha \blacktriangleleft_\alpha \Box &\in ECtx_\varepsilon^\alpha \end{aligned}$$

We write $E_B^A[M]$ for the term obtained by filling the hole $\Box$ in E_B^A by M.

Now we define staged reduction using the redex and evaluation contexts.

Definition 5 (Staged Reduction). *The staged reduction relation, written* $M \longrightarrow_s N$*, is defined by the least relation closed under the rules below.*

$$\begin{aligned} E^A_\varepsilon[(\lambda x : \tau.M)\ v^\varepsilon] &\longrightarrow_s E^A_\varepsilon[M[x \mapsto v^\varepsilon]] \\ E^A_\varepsilon[(\Lambda\alpha.v^\varepsilon)\ A] &\longrightarrow_s E^A_\varepsilon[v^\varepsilon[\alpha \mapsto A]] \\ E^A_\alpha[\blacktriangleleft_\alpha \blacktriangleright_\alpha v^\alpha] &\longrightarrow_s E^A_\alpha[v^\alpha] \end{aligned}$$

This reduction relation reduces a term in a deterministic, left-to-right, call-by-value manner. An application of an abstraction is executed only at stage ε and only a quotation at stage ε is spliced into the surrounding code—notice that, if $\blacktriangleright_\alpha v^\alpha$ is at stage ε, then the redex $\blacktriangleleft_\alpha \blacktriangleright_\alpha v^\alpha$ is at stage α. In other words, terms in brackets are not evaluated until the terms are run and arguments of a function are evaluated before the application. We show an example of staged reduction. Underlines show the redexes.

$$\begin{aligned} & (\Lambda\alpha.(\blacktriangleright_\alpha \underline{\blacktriangleleft_\alpha \blacktriangleright_\alpha((\lambda x : \mathrm{Int}.x)\ 10)}))\ \varepsilon \\ \longrightarrow_s & \underline{(\Lambda\alpha.(\blacktriangleright_\alpha((\lambda x : \mathrm{Int}.x)\ 10)))\ \varepsilon} \\ \longrightarrow_s & \underline{(\lambda x : \mathrm{Int}.x)\ 10} \\ \longrightarrow_s & 10 \end{aligned}$$

4 Properties of λ^{MD}

In this section, we show the basic properties of λ^{MD}: preservation, strong normalization, confluence for full reduction, and progress for staged reduction.

The Substitution Lemma in λ^{MD} is a little more complicated than usual because there are eight judgment forms and two kinds of substitution. The Term Substitution Lemma states that term substitution $[z \mapsto M]$ preserves derivability of judgments. The Stage Substitution Lemma states similarly for stage substitution $[\alpha \mapsto A]$.

We let $\mathcal{J}$ stand for the judgments $K\ \mathrm{kind}@A$, $\tau :: K@A$, $M : \tau@A$, $K \equiv J@A$, $\tau \equiv \sigma@A$, and $M \equiv N : \tau@A$. Substitutions $\mathcal{J}[z \mapsto M]$ and $\mathcal{J}[\alpha \mapsto A]$ are defined in a straightforward manner. Using these notations, the two substitution lemmas are stated as follows:

We proved the next two leammas by simultaneous induction on derivations.

Lemma 1 (Term Substitution). *If* $\Gamma, z : \xi@B, \Delta \vdash_\Sigma \mathcal{J}$ *and* $\Gamma \vdash_\Sigma N : \xi@B$*, then* $\Gamma, (\Delta[z \mapsto N]) \vdash_\Sigma \mathcal{J}[z \mapsto N]$*. Similarly, if* $\vdash_\Sigma \Gamma, z : \xi@B, \Delta$ *and* $\Gamma \vdash_\Sigma N : \xi@B$*, then* $\vdash_\Sigma \Gamma, (\Delta[z \mapsto N])$*.*

Lemma 2 (Stage Substitution). *If* $\Gamma \vdash_\Sigma \mathcal{J}$*, then* $\Gamma[\beta \mapsto B] \vdash_\Sigma \mathcal{J}[\beta \mapsto B]$*. Similarly, if* $\vdash_\Sigma \Gamma$*, then* $\vdash_\Sigma \Gamma[\beta \mapsto B]$*.*

The following Inversion Lemma is needed to prove the main theorems. As usual [27], the Inversion Lemma enables us to infer the types of subterms of a term from the shape of the term.

Lemma 3 (Inversion).

1. *If* $\Gamma \vdash_\Sigma (\lambda x : \sigma.M) : \rho$ *then there are* σ' *and* τ' *such that* $\rho = \Pi x : \sigma'.\tau'$, $\Gamma \vdash_\Sigma \sigma \equiv \sigma'@A$ *and* $\Gamma, x : \sigma'@A \vdash_\Sigma M : \tau'@A$.
2. *If* $\Gamma \vdash_\Sigma \blacktriangleright_\alpha M : \tau@A$ *then there is* σ *such that* $\tau = \triangleright_\alpha \sigma$ *and* $\Gamma \vdash_\Sigma M : \sigma@A$.
3. *If* $\Gamma \vdash_\Sigma \Lambda\alpha.M : \tau$ *then there is* σ *such that* $\sigma = \forall\alpha.\sigma$ *and* $\Gamma \vdash_\Sigma M : \sigma@A$.

Proof. Each item is strengthened by statements about type equivalence. For example, the first statement is augmented by

> If $\Gamma \vdash_\Sigma \rho \equiv (\Pi x : \sigma.\tau) : K@A$, then there exist σ' and τ' such that $\rho = \Pi x : \sigma'.\tau'$ and $\Gamma \vdash_\Sigma \sigma \equiv \sigma' : K@A$ and $\Gamma, x : \sigma@A \vdash_\Sigma \tau \equiv \tau' : J@A$.

and its symmetric version. Then, they are proved simultaneously by induction on derivations. Similarly for the others. □

Thanks to Term/Stage Substitution and Inversion, we can prove Preservation easily.

Theorem 1 (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 typing derivations. □

Strong Normalization is also an important property, which guarantees that no typed term has an infinite reduction sequence. Following standard proofs (see, e.g., [15]), we prove this theorem by translating λ^{MD} to the simply typed lambda calculus.

Theorem 2 (Strong Normalization). *If* $\Gamma \vdash_\Sigma M_1 : \tau@A$ *then there is no infinite sequence* $(M_i)_{i\geq 1}$ *of terms such that* $M_i \longrightarrow M_{i+1}$ *for* $i \geq 1$.

Proof. In order to prove this theorem, we define a translation $(\cdot)^\natural$ from λ^{MD} to the simply typed lambda calculus. Second, we prove the $\natural$-translation preserves typing and reduction. Then, we can prove Strong Normalization of λ^{MD} from Strong Normalization of the simply typed lambda calculus. □

Confluence is a property that any reduction sequences from one typed term converge. Since we have proved Strong Normalization, we can use Newman's Lemma [2] to prove Confluence.

Theorem 3 (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. We can easily show Weak Church-Rosser. Use Newman's Lemma. □

Now, we turn our attention to staged semantics. First, the staged reduction relation is a subrelation of full reduction, so Subject Reduction holds also for the staged reduction.

Theorem 4. *If $M \longrightarrow_s M'$, then $M \longrightarrow M'$.*

Proof. Easy. □

The following theorem Unique Decomposition ensures that every typed term is either a value or can be uniquely decomposed to an evaluation context and a redex, ensuring that a well-typed term is not immediately stuck and the staged semantics is deterministic.

Theorem 5 (Unique Decomposition). *If Γ does not have any variable declared at stage ε and $\Gamma \vdash_\Sigma M : \tau@A$, then either*

1. *$M \in V^A$, or*
2. *M can be uniquely decomposed into an evaluation context and a redex, that is, there uniquely exist B, E^A_B, and R^B such that $M = E^A_B[R^B]$.*

Proof. We can prove by straightforward induction on typing derivations. □

The type environment Γ in the statement usually has to be empty; in other words, the term has to be closed. The condition is relaxed here because variables at stages higher than ε are considered symbols. In fact, this relaxation is required for proof by induction to work.

Progress is a corollary of Unique Decomposition.

Corollary 1 (Progress). *If Γ does not have any variable declared at stage ε and $\Gamma \vdash_\Sigma M : \tau@A$, then $M \in V^A$ or there exists M' such that $M \longrightarrow_s M'$.*

5 Related Work

MetaOCaml is a programming language with quoting, unquoting, run, and CSP. Kiselyov [17] describes many applications of MetaOCaml, including filtering in signal processing, matrix-vector product, and a DSL compiler.

Theoretical studies on multi-stage programming owe a lot to seminal work by Davies and Pfenning [11] and Davies [10], who found Curry-Howard correspondence between multi-stage calculi and modal logic. In particular, Davies' λ° [10] has been a basis for several multi-stage calculi with quasi-quotation. λ° did not have operators for run and CSP; a few studies [3, 24] enhanced and improved λ° towards the development of a type-safe multi-stage calculus with quasi-quotation, run, and CSP, which were proposed by Taha and Sheard as constructs for multi-stage programming [31]. Finally, Taha and Nielsen invented the concept of environment classifiers [30] and developed a typed calculus λ^α, which was equipped with all the features above in a type sound manner and formed a basis of earlier versions of MetaOCaml. Different approaches to type-safe multi-stage programming with slightly different constructs for composing

and running code values have been studied by Kim, Yi, and Calcagno [16] and Nanevski and Pfenning [25].

Later, Tsukada and Igarashi [32] found correspondence between a variant of λ^{α} called $\lambda^{\triangleright}$ and modal logic and showed that run could be represented as a special case of application of a transition abstraction ($\Lambda\alpha.M$) to the empty sequence ε. Hanada and Igarashi [14] developed $\lambda^{\triangleright\%}$ as an extension $\lambda^{\triangleright}$ with CSP.

There is much work on dependent types and most of it is affected by the pioneering work by Martin-Löf [21]. Among many dependent type systems such as λ^{Π} [22], The Calculus of Constructions [9], and Edinburgh LF [15], we base our work on λLF [1] (which is quite close to λ^{Π} and Edinburgh LF) due to its simplicity. It is well known that dependent types are useful to express detailed properties of data structures at the type level such as the size of data structures [34] and typed abstract syntax trees [19,33]. The vector addition discussed in Section 3 is also such an example.

The use of dependent types for code generation is studied by Chlipala [8] and Ebner et al. [12]. They use inductive types to guarantee well-formedness of generated code. Aside from the lack of quasi-quotation, their systems are for heterogeneous meta-programming and compile-time code generation and they do not support features for run-time code generation such as run and CSP, as λ^{MD} does.

We discuss earlier attempts at incorporating dependent types into multi-stage programming. Pasalic and Taha [26] designed $\lambda_{H\circ}$ by introducing the concept of stage into an existing dependent type system λ_H [28]. However, $\lambda_{H\circ}$ is equipped with neither run nor CSP. Forgarty, Pasalic, Siek and Taha [13] extended the type system of MetaOCaml with indexed types. With this extension, types can be indexed with a Coq term. Chen and Xi [7] introduced code types augmented with information on types of free variables in code values in order to prevent code with free variables from being evaluated. These systems separate the language of type indices from the term language. As a result, they do not enjoy full-spectrum dependent types but are technically simpler because there is no need to take stage of types into account. Brady and Hammond [5] have discussed a combination of (full-spectrum) dependently typed programming with staging in the style of MetaOCaml to implement a staged interpreter, which is statically guaranteed to generate well-typed code. However, they focused on concrete programming examples and there is no theoretical investigation of the programming language they used.

Berger and Tratt [4] gave program logic for Mini-ML$_e^{\Box}$ [11], which would allow fine-grained reasoning about the behavior of code generators. However, it cannot manipulate open code which ours can deal with.

6 Conclusion

We have proposed a new multi-stage calculus λ^{MD} with dependent types, which make it possible for programmers to express finer-grained properties about the

behavior of code values. The combination leads to augmentation of almost all judgments in the type system with stage information. CSP and type equivalence (specially tailored for CSP) are keys to expressing dependently typed practical code generators. We have proved basic properties of λ^{MD}, including preservation, confluence, strong normalization for full reduction, and progress for staged reduction.

Developing a typechecking algorithm for λ^{MD} is left for future work. We expect that most of the development is straightforward, except for implicit CSP at the type-level and %-erasing equivalence rules.

Acknowledgments.

We would like to thank John Toman, Yuki Nishida, and anonymous reviewers for useful comments.

References

1. Aspinall, D., Hofmann, M.: Dependent types. In: Pierce, B.C. (ed.) Advanced topics in types and programming languages, chap. 2. MIT press (2005)
2. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press (1998)
3. Benaissa, Z.E.A., Moggi, E., Taha, W., Sheard, T.: Logical modalities and multi-stage programming. In: Federated logic conference (FLoC) satellite workshop on intuitionistic modal logics and applications (IMLA) (1999)
4. Berger, M., Tratt, L.: Program Logics for Homogeneous Generative Run-Time Meta-Programming. Logical Methods in Computer Science **Volume 11, Issue 1** (Mar 2015)
5. Brady, E., Hammond, K.: Dependently typed meta-programming. In: Trends in Functional Programming (2006)
6. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: International Conference on Generative Programming and Component Engineering. pp. 57–76. Springer (2003)
7. Chen, C., Xi, H.: Meta-programming through typeful code representation. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming. pp. 275–286. ICFP '03, ACM, New York, NY, USA (2003)
8. Chlipala, A.: Ur: Statically-typed metaprogramming with type-level record computation. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 122–133. PLDI '10, ACM, New York, NY, USA (2010)
9. Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. **76**(2-3), 95–120 (Feb 1988)
10. Davies, R.: A temporal-logic approach to binding-time analysis. In: Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on. pp. 184–195. IEEE (1996)
11. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM **48**(3), 555–604 (2001)
12. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. PACMPL **1**(ICFP), 34:1–34:29 (2017)

13. Fogarty, S., Pasalic, E., Siek, J., Taha, W.: Concoqtion: Indexed types now! In: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. pp. 112–121. PEPM '07, ACM, New York, NY, USA (2007)
14. Hanada, Y., Igarashi, A.: On cross-stage persistence in multi-stage programming. In: Codish, M., Sumii, E. (eds.) Functional and Logic Programming. pp. 103–118. Springer International Publishing, Cham (2014)
15. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM (JACM) **40**(1), 143–184 (1993)
16. Kim, I., Yi, K., Calcagno, C.: A polymorphic modal type system for lisp-like multi-staged languages. In: Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003). pp. 257–268 (2006)
17. Kiselyov, O.: Reconciling Abstraction with High Performance: A MetaOCaml approach. NOW Publishing (2018)
18. Kiselyov, O.: The design and implementation of BER metaocaml - system description. In: Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings. pp. 86–102 (2014)
19. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: Proc. of the 2nd Conference on Domain-Specific Languages (DSL '99). pp. 109–122 (1999)
20. Mainland, G.: Explicitly heterogeneous metaprogramming with metahaskell. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. pp. 311–322. ICFP '12, ACM, New York, NY, USA (2012)
21. Martin-Löf, P.: An intuitionistic theory of types: Predicative part. Logic Colloquium '73 **80** (01 1973)
22. Meyer, A.R., Reinhold, M.B.: "Type" is not a type. In: Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 287–295. POPL '86, ACM, New York, NY, USA (1986)
23. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**, 348–375 (1978)
24. Moggi, E., Taha, W., Benaissa, Z.E.A., Sheard, T.: An idealized MetaML: Simpler, and more expressive. In: Proc. of European Symposium on Programming (ESOP'99). Lecture Notes in Computer Science, vol. 1576, pp. 193–207 (1999)
25. Nanevski, A., Pfenning, F.: Staged computation with names and necessity. J. Funct. Program. **15**(5), 893–939 (2005)
26. Pasalic, E., Taha, W., Sheard, T.: Tagless staged interpreters for typed languages. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. pp. 218–229. ICFP '02, ACM, New York, NY, USA (2002)
27. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
28. Shao, Z., Saha, B., Trifonov, V., Papaspyrou, N.: A type system for certified binaries. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 217–232. POPL '02, ACM, New York, NY, USA (2002)
29. Taha, W.: A gentle introduction to multi-stage programming, part II. In: Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers. pp. 260–290 (2007)
30. Taha, W., Nielsen, M.F.: Environment classifiers. In: Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03). pp. 26–37. ACM, New York, NY, USA (2003)
31. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1-2), 211–242 (Oct 2000)

32. Tsukada, T., Igarashi, A.: A Logical Foundation for Environment Classifiers. Logical Methods in Computer Science **6**(4) (Dec 2010)
33. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03). pp. 224–235 (2003)
34. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98). pp. 249–257 (1998)