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 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 run for execution of programs dynamically generated by the code manipulation. Dependent types are expected to bring to multi-stage programming enforcement of strong invariants—beyond simple type safety—on the behavior of dynamically generated code. An extension is not trivial, however, because a type system would have to take stages—roughly speaking, the number of surrounding quotations—of types 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 subject reduction, confluence, strong normalization for full reduction, and progress [? – AI] for evaluation semantics. Motivated by code generators such that the type of generated code depends on a value from an earlier stage, we argue the significance of cross-stage persistence in dependently typed multi-stage programming and certain type equivalence that are not directly derived from reduction rules.

Keywords: Multistage programming · Dependent type

181 words. The abstract should briefly summarize the contents of the paper in 150–250 words. コードとかはボールドを使ったほうが良いかもしれない。Suspicious senetences are colored red. Multistage Programming と Multistage Calculus をどう使い分けるか?

1 Introduction

1.1 Multistage Programming

Multistage programming enables us to generate and run fragments of code at runtime. MetaOCaml[11] provide features for multistage programming, which are quote, unquote, and running code. Quote, written `.< e >.` in MetaOCaml, makes a code value from `e`. Run, written `run e` in MetaOCaml, runs a code

* Supported by organization x.

value of `e` and restore `e`. Unquote, written `.~ e` in MetaOCaml, expands a code value of `e`. Unlike run, unquote is supposed to use in a quoted fragment of code, that is, unquote cannot be used as an alternative of run. For example, the following MetaOCaml expression

```
let plusone = .< fun x -> x + 1 >. in .< .~plusone 2 >.
```

evaluates to `.< (fun x -> x + 1) 2 >.`. We can get the result with run.

```
run (let plusone = .< fun x -> x + 1 >. in .< .~plusone 2 >.)
```

is reduced to 3.

Cross-stage Persistence (CSP) is another primitive of multistage programming, which enable us to embed computed values into a code value. This is an example of CSP.

```
let plusone = fun x -> x + 1 in
let a = plusone 2 in  .< a >.
```

This program evaluates to `.< 3 >.` not `.< plusone 2 >.`. This is because the variable `a` is introduced into a code fragment using CSP. Therefore, the variable `a` is embedded after `a` is calculated. CSP is very important when we write practical programs because CSP enables us to use library functions in code fragments.

The main application of multistage calculi is program optimization. A famous example is `power` function. `power` takes two arguments, which are base and exponent. We can make specialized `power` functions for given exponents with multistage calculi and optimize them by unrolling a loop in functions for given exponents. power の実際のコードを持ってくるか?

Multistage programming can also optimize vector calculation. For example, `add` function, which takes two vectors and return the sum of them, is implemented with a loop in many cases. We can unroll `add` function for a given vector length and optimize it with multistage programming. MetaOCaml で書いた長さ付きベクトルの例?

Although unrolled `add` function is optimized, we cannot use it for different vector length. For example, when you optimize for the length of 5, you shouldn't use it for vectors of 3 lengths. Otherwise, we will get a Segmentation Fault error. This problem is serious but existing type systems for multistage calculi cannot prevent it.

1.2 Dependent Types

Dependent types are types which is dependent on values. We can use dependent types for securer programming than ordinary types. For example, we can realize vectors with their sizes with them.

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

`Vector` is a type constor which takes the length of vectors. `Vector 3` is a type for vectors whose lengths are 3. We can confirm the arguments of `add` function is the same with dependent types. secure programming? finer grained typing? とか

1.3 Multistage Programming with Dependent Types

この section は短いので上とくっつけてもいいかもしれない

As above, functions generated with multistage programming can take more restricted values as arguments. When you optimize `add` function for the length of 5, you should it only for 5 length vectors. We introduce dependent types into a multistage calculus so that the type system can guarantee optimized functions are used properly.

1.4 Organization of the Paper

The organization of this paper is the following. Section 2 gives an informal overview of λ^{MD}. Section 3 defines the syntax, type system, full reduction, evaluation contexts of λ^{MD}. Section 4 shows the properties of λ^{MD} including Unique Decomposition. Section 5 discusses future works. Finally, Section 6 discusses related works.

2 Informal Overview of λ^{MD}

In this section, we check λ^{MD} informally after checking $\lambda^{\triangleright\%}$ and λLF which are the basis of λ^{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 τ type. The type of $\blacktriangleleft_\alpha\ M$ is τ 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 β-reduction, there is a 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 λ^{MD}. この文はここに来るべきなのか? LTP の説明ではないが

The subscription α is a transition variable which is called a stage variable in λ^{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. run twice は表現が微妙

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

LTP に合わせて $\vdash^A$ を使うべきか? それとも LMD に合わせて $\vdash @A$ を使うべきか?

A type judgement of $\lambda^{\triangleright\%}$ is of the form $\Gamma \vdash M : \tau @ A$. Transition A means in which transition M exists. Therefore, transitions appear in typing rules, too.

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

T-$\blacktriangleright$, corresponding to quoting, means if M is typed τ at transition $A\alpha$ then $\blacktriangleright_\alpha\ M$, quoted M, is typed $\triangleright_\alpha \tau$ at A. T-$\blacktriangleleft$is converse of T-$\blacktriangleright$.

There are abstractions for transition variables and applications for transition abstraction in $\lambda^{\triangleright\%}$. They look like $\Lambda\alpha.M$ and $M\ A$, respectively. A transition abstraction binds a transition variable in a term. For example, all α 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 the 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 ϵ transition in λ^{MD} to simplify the system. この文はここに来るべきなのか? LTP の説明ではないが

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 ϵ. 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 enables us to embed value at an outer transition into an inner transition. inner / outer は適切か? $\%$ is dedicated to 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 arises when program residualization is used with CSP. Transition variables are classified into two kinds in $\lambda^{\triangleright\%}$ in order to deal with this difficulty.

2.2 λLF

λLF is a simple system of dependent types introduced in [1]. system / type system / calculus? It is based on Edinburgh LF[8]. Therefore, all constants and base types are declared in the signature. The λLF type theory generalizes simply typed lambda calculus by replacing the function type $\tau \to \sigma$ with the dependent product type $\Pi x : \tau.\sigma$.

In addition to ordinary typing rules like simply typed lambda calculus, there are kinding rules, well-formed kinding rules, term equivalence rules, type equivalence rules, and kind equivalence rules in λLF. Kinding rules and well-formed

kinding rules are introduced in order to prohibit making illegal types such as Vect Bool. For a well-formed type τ, $\Gamma \vdash \tau :: K$ means that τ has a kind K under the environment Γ and for a well-formed kind K, $\Gamma \vdash K$ means that K is a well-formed kind under an environment Γ.

Type equality rules are needed because the type equivalence is not obvious unlike simply typed lambda calculus. For example, Vect 7 should be equivalent to Vect $(3 + 4)$ but they are not equivalent seemingly. Thus, we must define equivalence rules. In λLF, equivalence is expressed with a symbol of $\equiv$. $\Gamma \vdash M \equiv N$ means a term M and a term N are equivalent under the environment Γ. $\Gamma \vdash \tau \equiv \sigma$ means a type τ and a type σ are equivalent under the environment Γ. $\Gamma \vdash K \equiv J$ means a kind K and a kind J are equivalent under the environment Γ.

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

Next, we develop λ^{MD} by extending $\lambda^{\triangleright\%}$ with λLF-like dependent types. From here, we use the word "stage" instead of "transition" because we develop a multi-stage calculus, not a multi-transition calculus. stage のほうが言葉としてふさわしいと言いたい There are three technical points in the extension from $\lambda^{\triangleright\%}$ to λ^{MD}.

First, the way of handling of constants and base types is the difference between λ^{MD} and $\lambda^{\triangleright\%}$. We adopt a signature Σ to handle constants and base types. This is because a signature simplifies kinding rules relating to type variables. A signature Σ is composed of pairs of a base type and its kind or a constant and its type. For example, if you want to use boolean, $\Sigma = \mathrm{Bool} :: *, \mathrm{true} : \mathrm{Bool}, \mathrm{false} : \mathrm{Bool}$ 具体的な導出例を出したほうがよいか? また、なぜ simple になるのを書くべきか?

Second, we need kinding rules and well-formed kinding rules in order to extend λ^{MD} with dependent types. It was lucky that almost all rules are determined easily. This is because multistage calculus and dependent types are almost orthogonal. orthogonal は抽象的すぎるか? Therefore, we can get kinding rules and well-formed kinding rules of λ^{MD} just by attaching stage anotations to ones of λLF. For example, K-Abs-LF is a kinding rule for a dependent type in λLF and K-Abs is a corresponding one.

$$\frac{\Gamma \vdash \tau :: * \qquad \Gamma, x : \tau @ A \vdash \sigma :: J}{\Gamma \vdash (\Pi x : \tau.\sigma) :: (\Pi x : \tau.J)}\text{K-Abs-LF}$$

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

Third, we also need type equivalence rules in λ^{MD} as a dependent type system. Although there are new primitives relating to stages which aren't in λLF, we can design all rules except Q-% easily. design? この段落が短い。後ろとくっつけてもいいが、段落の趣旨がボケる。

The biggest problem in extension is handling the CSP symbol $\%_\alpha$ in λ^{MD}. The equivalence simple rule to handle CSP is Q-Csp.

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

However, this rule isn't enough when the parameters of dependent types are cross-staed. We will discuss this problem in a later section and solve this problem with the new rule Q-%.

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

この段落で Q-Csp の持つ問題点を指摘すべきか? ただ、ここで mulmat の例を使うと 3 章と被る。

3 Formal Definition of $\lambda^{\mathbf{MD}}$

In this section, we present λ^{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 λ^{MD}.

3.1 Syntax

λ^{MD} is defined by the following grammar.

Type variables	X, Y, Z
Constants	c
Variables	x, y, z
Stage variables	α, β, γ
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 $\alpha, \beta,$ andγ range over the transition variables. A transition, denoted by A and B, is a finite sequence of transition variables. ϵ is a symbol for an empty transition.

A kind is $*$ or $\Pi x : \tau.K$. Types of terms have $*$ kinds and dependent types have Π 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 α-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 τ. An α-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 ϵ 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 λLF-like system, constants and type variables for base types are given in the signature Σ. For example, when we use integers in λ^{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 α is bound in $\Lambda\alpha.M$. We identify α-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 λ^{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 λ^{MD}. Congruence rules which are omitted from the definition.*

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

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 ϵ reduces to the body of abstraction where ϵ is substituted to the stage variable. Application of a non-ϵ stage to a stage abstraction is prohibited in order to simplify λ^{MD}. There is no reduction rule for CSP as with Hanada and Igarashi [7]. $\%_\alpha$, the symbol for CSP, is disappeared when ϵ is substituted to α. $\blacktriangleright_\alpha$ and $\blacktriangleleft_\alpha$ is disappeared in the same way as $\%_\alpha$.

3.3 Type System

In this section, we define the type system of λ^{MD}. λ^{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 λ^{MD}. We show them in Figure 1.

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

Fig. 1. Six types of judgement in λ^{MD}

Especially, we discuss on the type equality of λ^{MD} in detail because it is the main contribution of this paper. λ^{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 λ^{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, τ, has the proper kind for terms, $*$. T-Abs also checks the kind of the type.

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

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

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 ϵ stage to a stage abstraction, and cross-stage persistence, respectively. Please check Hanada and Igarashi [7] and Tsukada and Igarashi [22] 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.

Type Equivalence Rules

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

When we design type equivalence rules of λ^{MD}, there are two design choices. One is defining type equivalence by β-equality after we define β-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.

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 Π 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-▶, Q-◀, 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

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

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

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

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

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

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

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

Fig. 3. Typing Rules

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

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

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

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

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

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

Fig. 4. Type Equivalence Rules

equivalence relationship as with the type equivalence. The rule Q-β 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.

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. Term Equivalence Rules

The rule Q-%is something special because it has no background of syntax or reductions. But we add this rule to λ^{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 $\triangleright_\alpha \Pi z : \mathrm{Int}.(\mathrm{Mat}\ z\ \%_\alpha 5) \to (\mathrm{Mat}\ \%_\alpha 5\ \%_\alpha 3) \to (\mathrm{Mat}\ z\ \%_\alpha 3)$. But this type is difficult 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 λ^{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 λ^{MD} interpreter. In this reduction, $\longrightarrow_\beta$ and $\longrightarrow_\Lambda$ are allowed only at stage ϵ and $\longrightarrow_\blacklozenge$ is allowed only at stage α.

Definition 3 (Values). *The family V^A of sets of values, ranged over by v^A and the sets of ϵ-redexes (ranged over by R^ϵ) and α-redexes (ranged over by R^α) are defined by following grammar. In the grammar $A \neq \epsilon$.*

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

Values at ϵ stage are a λ abstraction, a quoted code, or a Λ abstraction. The body of a λ abstraction can be any term but the body of Λ abstraction only can be a value. This restriction means that the body of *Lambda* abstraction must be evaluated before application of an ϵ stage. Values at non-ϵ 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.) $A \neq \epsilon$*

$$E^\epsilon_B \in ECtx^\epsilon_B ::= \Box \ (\mathit{if}\ B = \epsilon) \mid E^\epsilon_B\ M \mid v^e\ E^\epsilon_B \mid \blacktriangleright_\alpha\ E^\alpha_B \mid \Lambda\alpha.E^\epsilon_B \mid E^\epsilon_B\ \epsilon$$
$$E^A_B \in ECtx^A_B ::= \Box \ (\mathit{if}\ A = B) \mid \lambda x : \tau.E^A_B \mid E^A_B\ M \mid v^A\ E^A_B \mid E^\epsilon_B \mid \blacktriangleright_\alpha\ E^{A\alpha}_B \mid \blacktriangleleft_\alpha\ E^{A'}_B \ (\mathit{where}\ A'\alpha = A)$$
$$\mid \Lambda\alpha.E^\epsilon_B \mid E^A_B\ \epsilon \mid \%_\alpha\ E^{A'}_B \ (\mathit{where}\ A'\alpha = A)$$

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.*

$$E^A_\epsilon[(\lambda x : \tau.M)\ v^\epsilon] \longrightarrow_s E^A_\epsilon[M[x \mapsto v^\epsilon]]$$
$$E^A_\epsilon[(\Lambda\alpha.v^\epsilon)\ \epsilon] \longrightarrow_s E^A_\epsilon[v^\epsilon[\alpha \mapsto \epsilon]]$$
$$E^A_\alpha[\blacktriangleleft_\alpha \blacktriangleright_\alpha\ v^\alpha] \longrightarrow_s E^A_\alpha[v^\alpha]$$

Fig. 6. Staged Reduction

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

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

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

Theorem 1 (Substituition Lemma).

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

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 Π 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 Λ 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 λ^{MD} to Simply Typed Lambda Calculus. Second, we prove $\natural$ preserves typing and β-reductions. Then, we can prove Strong Normalization of λ^{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 λ^{MD}, we can use Newman's lemma to prove Confluence of λ^{MD}. Then, what we must show is Weak Church-Rosser Property now. When we consider two different redexes in a λ^{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 λ^{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 Works

There are many papers on dependent types and most of them are affected with the pioneering work of Per Martin-Löf. λ^{Π}[16], Calculus of Constructions[4], and Harper, Honsell and Plotkin[8] are famous papers on dependent types. In Advanced Topics in Types and Programming Languages[1], a dependent type system λLF is designed with the basis of [8]. We adopted [1]-like dependent types for λ^{MD}.

One can use dependent types in programming languages such as Idris[2] or interactive theorem provers such as Coq[21] which based on [4]. この段落要らない or 後ろとくっつけたほうが良いかも知れない?

Applications of dependent types for real problems were also researched. In Xi and Pfenning[25], they extended SML with restricted dependent types and succeeded in reducing the bounds checking of arrays. Xi also gave dead code elimination and loop unrolling as examples of dependent types application in [26].

Besides, multistage calculi also have a long history of research. Davis revealed there is Curry-Howard correspondence between a multistage calculus and modal logic in λ°[5]. However, λ° don't have operators for run and CSP. run は bold か? Besides より適切な接続詞はないのか?

Taha and Sheard introduced run and CSP to a multistage calculus in [20]. Additionally, Taha and Nielsen invented the concept of environment classifiers in λ^α[19] and construct the type system for λ^α, which can handle quoting, unquoting, run, and CSP. In $\lambda^\triangleright$[22], Tsukada and Igarashi found modal logic which corresponds to a multistage calculus with environment classifiers and show that

run can be represented as application of ϵ to transition abstractions. In $\lambda^{\triangleright\%}$ [7], Hanada and Igarashi extended $\lambda^{\triangleright}$ with CSP and discuss code residualization which allows us to dump the quoted code into an external file.

MetaOCaml is a programming language with quoting, unquoting, run, and CSP. Oleg gave many examples of application of MetaOCaml in [10], which include filtering in signal processing, matrix-vector product, and DSL compiler.

- On Cross-Stage Persistence in Multi-Stage[7]
  CSP も入りました
- Eliminating Array Bound Checking Through Dependent Types[25]
- MetaML and Multi-stage Programming with Explicit Annotations[20]
- Idris, a general-purpose dependently typed programming language: Design and implementation[2]
- A Logical Foundation for Environment Classifiers[22]
- Environment classifiers[19]
- A framework for defining logics[8]
  Σ の使い方を確認した
- The Design and Implementation of BER MetaOCaml - System Description[11]
- Refined Environment Classifiers[12]
- Staging with control: type-safe multi-stage programming with control operators[17]
- Partial evaluation and automatic program generation [9]
- Efficient multi-level generating extensions for program specialization [6]
- Dependent types in practical programming[26]
  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[24]
  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 [23]
- C and tcc: a language and compiler for dynamic code generation [18]
- Run-time bytecode specialization [15]
- A tour of Tempo: A program specializer for the C language [3]
- Optimizing ML with run-time code generation [13]
- Efficient incremental run-time specialization for free [14]

6 Conclusions

References

1. Aspinall, D., Hofmann, M.: Advanced topics in types and programming languages, chap. 2 Dependent Types. MIT press (2005)
2. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23**(5), 552–593 (2013)

3. 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)
4. Coquand, T., Huet, G.: The calculus of constructions. Tech. Rep. RR-0530, INRIA (May 1986), https://hal.inria.fr/inria-00076024
5. 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)
6. 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)
7. Hanada, Y., Igarashi, A.: On Cross-Stage Persistence in Multi-Stage pp. 103–118 (2014)
8. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM (JACM) **40**(1), 143–184 (1993)
9. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Peter Sestoft (1993)
10. Kiselyov, O.: Reconciling Abstraction with High Performance: A MetaOCaml approach. now (2018), https://ieeexplore.ieee.org/document/8384206
11. 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
12. Kiselyov, O., Kameyama, Y., Sudo, Y.: Refined environment classifiers. In: Asian Symposium on Programming Languages and Systems. pp. 271–291. Springer (2016)
13. Lee, P., Leone, M.: Optimizing ML with run-time code generation, vol. 31. ACM (1996)
14. Marlet, R., Consel, C., Boinot, P.: Efficient incremental run-time specialization for free. In: ACM SIGPLAN Notices. vol. 34, pp. 281–292. ACM (1999)
15. Masuhara, H., Yonezawa, A.: Run-time bytecode specialization. In: Symposium on Program as Data Objects. pp. 138–154. Springer (2001)
16. 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). https://doi.org/10.1145/512644.512671, http://doi.acm.org/10.1145/512644.512671
17. 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)
18. 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)
19. Taha, W., Nielsen, M.F.: Environment classifiers. In: ACM SIGPLAN Notices. vol. 38, pp. 26–37. ACM (2003)
20. 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
21. The Coq Development Team: The coq proof assistant reference manual (2009)

22. 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
23. Wickline, P., Lee, P., Pfenning, F.: Run-time code generation and Modal-ML, vol. 33. ACM (1998)
24. Xi, H., Harper, R.: A dependently typed assembly language. In: ACM SIGPLAN Notices. vol. 36, pp. 169–180. ACM (2001)
25. 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
26. 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.