Master's Thesis

A Dependently Typed Multi-Stage Calculus

Supervisor Professor Atsushi Igarashi

Department of Communications and Computer Engineering
Graduate School of Informatics
Kyoto University

Akira Kawata

February 15, 2020

A Dependently Typed Multi-Stage Calculus

Akira Kawata

Abstract

Multi-stage programming is a programming paradigm that enables generation and evaluation of code at run-time. For instance, MetaOCaml, which is a well-known multi-stage programming language supporting quasi-quotation and cross-stage persistence for the manipulation of code fragments as first-class values and an evaluation construct for the execution of programs dynamically generated by code manipulation. A significant merit of multi-stage programming is the utilization of run-time information during code generation. For example, we can generate an efficient vector sum function for arbitrary fixed-length vectors by unrolling loops.

However, such optimization may cause severe problems because specialized functions only can be used with restricted arguments. In the case of a vector addition function, the generated function is specialized for the given length, and must be used with vectors of the correct length. Otherwise, the function can access outside of vectors because the loop-unrolled code never checks the boundaries of its input.

To resolve the above problem, we study a dependently typed extension of a multi-stage programming language à la MetaOCaml. Dependent types bring to multi-stage programming the enforcement of strong invariants—beyond simple type safety—on the behavior of dynamically generated code. In the case of the vector sum function, dependent types restrict the input arguments to vectors of the correct length. However, extending multi-stage programming with dependent types is non-trivial because such a type system must integrate stages of types, which is the number of surrounding quotations.

To rigorously study properties of such an extension, we develop λ^{MD}, which is an extension of typed calculus $\lambda^{\triangleright\%}$ proposed by Hanada and Igarashi with dependent types, and prove its properties, including preservation, confluence, strong normalization for full reduction, and progress for staged reduction. Furthermore, we design algorithmic typing to implement a type-inference program and prove its equivalence to the original typing.

依存型付多段階計算体系

河田 旺

内容梗概

多段階計算は実行時におけるコードの生成，評価を可能にするプログラミング手法である．MetaOCaml は多段階計算を利用可能なプログラミング言語であり，コードの生成，埋め込み，多段階埋め込み (Cross-Stage Persistence)，コードの実行といった他段階計算の主要な機能が利用できる．多段階計算の主な利点は実行時の情報をコードの生成に利用できることである．例として 2 つのベクトルの和を計算する関数を考える．このときベクトルの長さの情報が利用できれば関数内のループを展開することで関数を高速化できる．

しかし，このような高速化には，高速化した関数が限定された引数しか受け取れないという弊害もある．ベクトル和を計算する関数の場合，生成された関数は特定の長さに特殊化されており，その以外の長さのベクトルに対して適用することはできない．適用した場合，関数はベクトルの範囲外にアクセスしているかどうかを検査せずにベクトルの範囲外のメモリ領域にアクセスしてしまう可能性がある．

このような問題を解決するために，我々は MetaOCaml 風の依存型付多段階計算体系を提案する．依存型を使えば多段階計算を扱うプログラムにコードの振る舞いに対する単純型より強い不変条件を与えることができる．ベクトル和を求める関数の場合，引数の長さは依存型によって適切なものだけに制限することができる．しかしステージ，つまり式の周りの引用符の数，を型システムに組み込む必要がありため，依存型による多段階計算体系の拡張は自明ではなかった．

このような計算体系の性質を厳密に研究するため，我々は Hanada, Igarashi による $\lambda^{\triangleright\%}$ を拡張し λ^{MD} を提案する．そしてこの体系について保存性，合流性，full reduction に対する強正規化性とステージを考慮した簡約における進行性を示した．更に，我々は λ^{MD} の型推論器を設計するのに必要な algorithmic typing も設計し，その規則が元の型付け規則と同値であることを示した．

A Dependently Typed Multi-Stage Calculus

Contents

Chapter 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 [1–3].

MetaOCaml [4, 5] is an extension of OCaml[1)] with special constructs for multi-stage programming, including brackets and escape, which are (hygienic) quasi-quotation, and `run`, which is similar to `eval` in Lisp, and cross-stage persistence (CSP) [6]. 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 [7] 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 [8], is not strong enough to guarantee invariant beyond simple types. For example, Kiselyov [1] 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

[1)] http://ocaml.org

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 [9]. 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 $\lambda^{\triangleright\%}$ [10] 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} [7], which formalizes the design of MetaOCaml more faithfully. Dependent types are based on λLF [11], 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.

- We give algorithmic typing for λ^{MD} which is essential to implement a type checker and show the equivalence between algorithmic typing and normal typing.

The combination of multi-stage programming and dependent types has been discussed by Pasalic, Taha, and Sheard [12] and Brady and Hammond [13] 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.

1.3.1 Organization of the Thesis

The organization of this paper is as follows. Section Chapter 2 gives an informal overview of λ^{MD}. Section Chapter 3 defines λ^{MD} and Section Chapter 4 shows properties of λ^{MD}. Section Chapter 5 discusses related work and Section Chapter 6 concludes the paper with discussion of future work.

Chapter 2 Informal Overview of λ^{MD}

We describe our calculus λ^{MD} informally. λ^{MD} is based on $\lambda^{\triangleright\%}$ [10] 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*[1)] 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 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

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

[1)] In Hanada and Igarashi [10], it was called a *transition variable*, which is derived from correspondence to modal logic, studied by Tsukada and Igarashi [14].

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.[1)] 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 [11]—to $\lambda^{\triangleright\%}$. Types can be indexed by terms as in `Vector` in Section Chapter 1 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

[1)] In Hanada and Igarashi [10], it is written $\Gamma \vdash^A M : \tau$.

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 $x : \mathrm{Int}$ 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 $\mathrm{Vector} :: \mathrm{Int} \to *$ 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.

Chapter 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 $dom(\Sigma)$ and $dom(\Gamma)$ for the set of (type-level) constants and variables declared in Σ and Γ, respectively. As in other multi-stage calculi [7, 10, 14], 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 [10], 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.*

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

Figure 1: Example of reduction.

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

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

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 [10]. The CSP operator $\%_\alpha$ disappears when ε is substituted for α.

We show an example of a reduction sequence in Figure 1. Underlines show the redexes. Please notice that terms in type annotaions are also reduced like Figure 2.

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

$$\lambda v : (\mathrm{Vec}\ (\underline{1+3})).v$$
$$\longrightarrow_\beta \lambda v : (\mathrm{Vec}\ 4).v$$

Figure 2: Example of reduction in a type annotation.

$\vdash \Sigma$	signature well-formedness
$\vdash_\Sigma \Gamma$	type environment well-formedness
$\Gamma \vdash_\Sigma K\ \ \mathrm{kind} @A$	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

Figure 3: Eight judgment forms of the type system of λ^{MD}.

equivalence. We list the judgment forms in Figure 3. They are all defined in a mutual recursive manner. We will discuss each judgment below.

3.3.1 Signature and Type Environment Well-formedness

The rules for Well-formed signatures and type environments are shown below:

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

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

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.

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash_\Sigma M : \tau@A \qquad \Gamma \vdash_\Sigma \tau :: *@A\alpha}{\Gamma \vdash_\Sigma \%_\alpha M : \tau@A\alpha}\ (\text{T-Csp})$$

Figure 4: Typing Rules.

3.3.2 Kind Well-formedness and Kinding

The rules for kind well-formedness and kinding are a straightforward adaptation from λLF and $\lambda^{\triangleright\%}$. So, we omit the definitions.

3.3.3 Typing

The typing rules of λ^{MD} are shown in Figure 4. 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 Chapter 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 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 and type τ is also legitimate at stage $A\alpha$, 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 $\lambda^{\triangleright\%}$.

3.3.4 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 5, 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 well-typed at the next stage $A\alpha$. For example, constants do not depend on

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

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

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

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

Figure 5: Term Equivalence Rules.

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.

3.3.5 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:

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

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 $\triangleright_\alpha \mathrm{Vector}\ n$.)

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

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

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 $\forall\beta.\triangleright_\beta(\mathrm{Vector}\ (\%_\beta 5) \to \mathrm{Vector}\ (\%_\beta 5) \to \mathrm{Vector}\ (\%_\beta 5))$. 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.

Remark: 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 [10] 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.

3.4 Algorithmic Typing

Algorithmic typing rules are rules for the implementation of type inference program which takes an environment Γ, a signature Σ, a term M, and a stage A and returns a type τ which satisfies $\Gamma \vdash_\Sigma M : \tau@A$. If there is no such τ, the algorithm tells failure. We already defined typing rules for λ^{MD} in Section 3.3.3, but they include some non-syntax directed rules. Non-syntax directed rules mean their premise cannot be decided from the goal. When we implement the type inference program of λ^{MD}, it is necessary to make up type derivation trees from bottom to up. So, we need to make all the typing rules syntax-directed so that algorithmic typing is equivalent to the original typing. Using this algorithmic typing, we can make the a type inference algorithm easily just by applying the unique suitable rule one by one. We show the equivalence in Section Chapter 4.

In algorithmic typing, there are 8 judgments as with original typing in Figure 3, i.e., well-formed signature, well-formed environment, well-formed kinding, kinding, typing, kind equivalence, type equivalence, and term equivalence. We replace $\vdash_\Sigma$ with $\vdash\!\blacktriangleright$ in judgments to distinguish them from judgments of the original typing and omit kinds and types from typing equivalence and term equivalence, respectively. The definitions of well-formed signature, well-formed environment, and well-formed kinding are the same with original typing, so we don't explain them here.

Kinding and typing rules are almost the same as the original typing, but there are no conversion rules, such as T-Conv and K-Conv. Conversion rules

are not syntax-directed because we can apply them at any typing or kinding judgments. Thus, we must remove them from algorithmic typing. As you can see in [11], all uses of conversion rules in a dependent type system can be put into application rules. The simplification is possible because only application rules check the equivalence between the actual argument and formal argument, but all other rules don't check the equality of types. Therefore, the equivalence of types becomes a problem only in application rules. Then, if we replace type equality checks in application rules with a conversion check, we can eliminate conversion rules from algorithmic typing. The two application rules are as follows.

$$\frac{\Gamma \vdash\!\blacktriangleright_\Sigma \sigma :: (\Pi x : \tau.K)@A \qquad \Gamma \vdash\!\blacktriangleright_\Sigma M : \tau'@A \qquad \Gamma \vdash\!\blacktriangleright_\Sigma \tau' :: *@A \qquad \Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A}{\Gamma \vdash\!\blacktriangleright_\Sigma \sigma\ M :: K[x \mapsto M]@A}\ (\text{KA-App})$$

$$\frac{\Gamma \vdash\!\blacktriangleright_\Sigma M : (\Pi(x : \sigma).\tau) \qquad \Gamma \vdash\!\blacktriangleright_\Sigma N : \sigma'@A \qquad \Gamma \vdash\!\blacktriangleright_\Sigma \sigma' :: *@A \qquad \Gamma \vdash\!\blacktriangleright_\Sigma \sigma \equiv \sigma'@A}{\Gamma \vdash\!\blacktriangleright_\Sigma M\ N : \tau[x \mapsto N]@A}\ (\text{TA-App})$$

Unlike the original rules, the kind and type equivalence rules of the algorithmic typing have no rules for transitivity, symmetry, and reflexivity. These rules are omitted because they are non-syntax directed. Instead of including them in the rules, we prove transitivity, symmetry, and reflexivity as meta theorems.

Unlike all other rules of the algorithmic typing, the term equivalence rules of algorithmic typing are completely different from the original ones. We replaced all rules of term equivalence with QA-ANF. Although the original equivalence rules check the equivalence of terms by constructing a derivation tree, the algorithmic equivalence rules calculate the normal form of two terms and compare them with just α-equivalence. $\mathrm{ANF}(M)$ is the normal form of M under algorithmic reduction as defined in Figure 6. Algorithmic reduction is composed of normal reductions of λ^{MD}, which are (A-β), (A-$\blacktriangleleft\blacktriangleright$), and (A-$\Lambda$), and one special reduction (A-%), which corresponds to Q-% in the normal term equivalence rules. Just the same as the normal reduction $M \longrightarrow M'$, we define algorithmic reduction $M \longrightarrow_{\mathrm{A}}$.

$$\frac{\mathrm{ANF}(M) \equiv_\alpha \mathrm{ANF}(N)}{\Gamma \vdash\!\blacktriangleright M \equiv N@A}\ (\text{QA-ANF})$$

$$(\lambda x : \tau.M)N \longrightarrow_{\mathrm{A}} M[x \mapsto N] \qquad \text{(A-}\beta\text{)}$$

$$\blacktriangleleft_\alpha(\blacktriangleright_\alpha M) \longrightarrow_{\mathrm{A}} M \qquad \text{(A-}\blacktriangleleft\blacktriangleright\text{)}$$

$$(\Lambda\alpha.M)\ A \longrightarrow_{\mathrm{A}} M[\alpha \mapsto A] \qquad \text{(A-}\Lambda\text{)}$$

$$\%_\alpha M \longrightarrow_{\mathrm{A}} M \qquad (\emptyset = \mathrm{FV}(\%_\alpha M)) \qquad \text{(A-}\%\text{)}$$

Figure 6: Algorithmic Basic Reduction

3.5 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 [10], we introduce (small-step, call-by-value) staged semantics, where only β-reduction or Λ-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 [10] 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.

$$R^\varepsilon ::= (\lambda x : \tau.M)\ v^\varepsilon \mid (\Lambda\alpha.v^\varepsilon)\ \varepsilon$$
$$R^\alpha ::= \ \blacktriangleleft_\alpha \blacktriangleright_\alpha v^\alpha$$

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:

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

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.*

$$E_\varepsilon^A[(\lambda x : \tau.M)\ v^\varepsilon] \longrightarrow_s E_\varepsilon^A[M[x \mapsto v^\varepsilon]]$$
$$E_\varepsilon^A[(\Lambda\alpha.v^\varepsilon)\ A] \longrightarrow_s E_\varepsilon^A[v^\varepsilon[\alpha \mapsto A]]$$
$$E_\alpha^A[\blacktriangleleft_\alpha \blacktriangleright_\alpha v^\alpha] \longrightarrow_s E_\alpha^A[v^\alpha]$$

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.

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

Chapter 4 Properties of λ^{MD}

4.1 Properties of Typing

In this subsection, 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 a similar property for stage substitution $[\alpha \mapsto A]$.

We let $\mathcal{J}$ stand for the judgments $K\ \mathrm{kind}\,@A$, $\tau :: K@A$, $M : \tau@A$, $K \equiv K'@A$, $\tau \equiv \tau'@A$, and $M \equiv M' : \tau@A$. 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:

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* $\vdash_\Sigma \Gamma[\beta \mapsto B]$.

Proof. By simultaneous induction on derivations. □

We prove agreement to use later in the completeness of the algorithmic typing.

Lemma 3 (Agreement)**.**

- *If* $\Gamma \vdash_\Sigma \tau :: K@A$*, then* $\Gamma \vdash_\Sigma K\ \mathrm{kind}\,@A$.
- *If* $\Gamma \vdash_\Sigma M : \tau@A$*, then* $\Gamma \vdash_\Sigma \tau :: *@A$.
- *If* $\Gamma \vdash_\Sigma K \equiv J@A$*, then* $\Gamma \vdash_\Sigma K@A$ *and* $\Gamma \vdash_\Sigma J@A$.
- *If* $\Gamma \vdash_\Sigma \tau \equiv \sigma :: K@A$*, then* $\Gamma \vdash_\Sigma \tau :: K@A$ *and* $\Gamma \vdash_\Sigma \sigma :: K@A$.
- *If* $\Gamma \vdash_\Sigma M \equiv N : \tau@A$*, then* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $\Gamma \vdash_\Sigma N : \tau@A$.

Proof. By induction on the derivation tree. □

The following Inversion Lemma is needed to prove the main theorems. As usual [16], the Inversion Lemma enables us to infer the types of subterms of a

term from the shape of the term.

Lemma 4 (Inversion)**.**

- *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* $\Gamma, x : \sigma'@A \vdash_\Sigma M : \tau'@A$.
- *If* $\Gamma \vdash_\Sigma \blacktriangleright_\alpha M : \tau@A$ *then there is* σ *such that* $\tau = \triangleright_\alpha \sigma$ *and* $\Gamma \vdash_\Sigma M : \sigma@A$.
- *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 Type Preservation.

Lemma 5 (Type and Equivalence Preservation)**.**

- *If* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $M \longrightarrow M'$, *then* $\Gamma \vdash_\Sigma M' : \tau@A$.
- *If* $\Gamma \vdash_\Sigma M \equiv M' : \tau@A$ *and* $M \longrightarrow M''$, *then* $\Gamma \vdash_\Sigma M'' \equiv M' : \tau@A$.

Proof. First, there are three base 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. □

Theorem 1 (Type Preservation)**.** *If* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $M \longrightarrow M'$, *then* $\Gamma \vdash_\Sigma M' : \tau@A$.

Proof. This is a corollary of Lemma 5. □

Strong Normalization is also an important property, which guarantees that no typed term has an infinite reduction sequence. To prove this, we translate λ^{MD} to LF [15] preserving β-reductions.

Definition 6 ($\natural$ translation)**.** *The* $\natural$ *translation is a translation from* λ^{MD} *to LF.*

- *Kind*

$$\natural(*) = \text{Type}$$
$$\natural(\Pi x : \tau.K) = \Pi x : \natural(\tau).\natural(K)$$

- *Term*

$$\natural(c) = c$$
$$\natural(x) = x$$
$$\natural(\lambda x : \tau.M) = \lambda x : \natural(\tau).\natural(M)$$
$$\natural(M\ N) = \natural(M)\ \natural(N)$$
$$\natural(\blacktriangleright_\alpha M) = \natural(M)$$
$$\natural(\blacktriangleleft_\alpha M) = \natural(M)$$
$$\natural(\Lambda\alpha.M) = \natural(M)$$
$$\natural(M\ B) = \natural(M)$$

- *Type*

$$\natural(X) = X$$
$$\natural(\Pi x : \tau.\sigma) = \Pi x : (\tau).\natural(\sigma)$$
$$\natural(\tau\ M) = \natural(\tau)\natural(M)$$
$$\natural(\triangleright_\alpha \tau) = \natural(\tau)$$
$$\natural(\forall\alpha.\tau) = \natural(\tau)$$

- *Context*

$$\natural(\phi) = \phi$$
$$\natural(\Gamma, x : \tau@A) = \natural(\Gamma), \natural(x) : \natural(\tau)$$

- *Signature*

$$\natural(\phi) = \phi$$
$$\natural(\Sigma, c : \tau) = \natural(\Sigma), \natural(c) : \natural(\tau)$$
$$\natural(\Sigma, X : K) = \natural(\Sigma), \natural(X) : \natural(K)$$

We show $\natural$ translation preserves typing and equivalence of λ^{MD}.

Lemma 6 (Substitution and $\natural$)**.** *If* $\Gamma, x : \sigma\Delta \vdash_\Sigma M : \tau@A$ *and* $\Gamma \vdash_\Sigma N : \sigma@A$ *in* λ^{MD} *then* $\natural(M[x \mapsto N]) = \natural(M)[x \mapsto \natural(N)]$

Proof. By induction on the type derivation tree of $\Gamma, x : \sigma\Delta \vdash_\Sigma M : \tau@A$. □

Lemma 7 (Preservation of Judgements in $\natural$)**.** *There are following relations between judgments in* λ^{MD} *and LF.*

- *If* $\vdash \Sigma$ *then* $\natural(\Sigma)$ sig.
- *If* $\vdash_\Sigma \Gamma$ *then* $\vdash_{\natural(\Sigma)} \natural(\Gamma)$.
- *If* $\Gamma \vdash_\Sigma K$ kind $@A$ *then* $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(K)$.
- *If* $\Gamma \vdash_\Sigma \tau : K@A$ *then* $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(\tau) : \natural(K)$.
- *If* $\Gamma \vdash_\Sigma M : \tau@A$ *then* $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(M) : \natural(\tau)$.
- *If* $\Gamma \vdash_\Sigma K \equiv K'@A$ *then* $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(K) \equiv \natural(K')$, $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(K)$ *and*

$\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(K')$.

- *If* $\Gamma \vdash_\Sigma \tau \equiv \tau' : K@A$ *then* $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(\tau) \equiv \natural(\tau')$, $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(\tau) : \natural(K)$ *and* $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(\tau') : \natural(K)$.
- *If* $\Gamma \vdash_\Sigma M \equiv M' : \tau@A$ *then* $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(M) \equiv \natural(M')$, $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(M) : \natural(\tau)$ *and* $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(M') : \natural(\tau)$.

Proof. We can prove using induction on the type derivation tree. We show the case of T-App and T-Conv as examples. Other cases are easy.

Case T-App**:** $\Gamma \vdash_\Sigma M : (\Pi(x : \sigma).\tau)@A$ and $\Gamma \vdash_\Sigma N : \sigma@A$

From the induction hypothesis, we have $\natural(\Gamma) \vdash_{\natural(\sigma)} \natural(M) : \Pi x : \natural(\sigma).\natural(\tau)$ and $\natural(\Gamma) \vdash_{\natural(\sigma)} \natural(N) : \natural(\sigma)$. Using B-APP-OBJ rule in LF, we get
$\natural(\Gamma) \vdash_{\natural(\sigma)} \natural(M)\ \natural(N) : \natural(\tau)[x \mapsto \natural(N)]$. Because $\natural(M)\ \natural(N) = \natural(M\ N)$ from the definition of $\natural$ and $\natural(\tau)[x \mapsto \natural(N)] = \natural(\tau[x \mapsto N])$ from Lemma 6,
$\natural(\Gamma) \vdash_{\natural(\sigma)} \natural(M\ N) : \natural(\tau[x \mapsto N])$ in LF.

Case T-Conv**:** $\Gamma \vdash_\Sigma M : \tau@A$ and $\Gamma \vdash_\Sigma \tau \equiv \tau' : K@A$

By the induction hypothesis, $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(M) : \natural(\tau)$, $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(\tau) \equiv \natural(\tau')$, $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(\tau) : \natural(K)$ and $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(\tau') : \natural(K)$. From Lemma 3, K is $*$. Therefore, $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(\tau')$: Type. Now, we can use B-CONV-OBJ and get $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(M) : \natural(\tau')$. □

Lemma 8 (Preservation of β-Reduction in $\natural$)**.** *If* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $M \longrightarrow_\beta N$ *in* λ^{MD} *then* $\natural(M) \longrightarrow_\beta^+ \natural(N)$.

Proof. By induction on the derivation of β-reduction of λ^{MD}. We show only the main case.

Case $(\lambda x : \tau.M)\ N \longrightarrow_\beta M[x \mapsto N]$**:**

From the definition of $\natural$, $\natural((\lambda x : \tau.M)\ N) = \lambda x : \natural(\tau).\natural(M)\ \natural(N)$. Because $\lambda x : \natural(\tau).\natural(M)\ \natural(N)$ is a typed term in LF by Lemma 7, we can perform β-reduction from it. As a result of the reduction, we get $\natural(M)[x \mapsto \natural(N)]$. From 6, $\natural(M[x \mapsto N]) = \natural(M)[x \mapsto \natural(N)]$. □

Lemma 9 (Preservation of Λ-Reduction in $\natural$)**.** *If* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $M \longrightarrow_\Lambda N$ *in* λ^{MD} *then* $\natural(M) = \natural(N)$.

Proof. We prove by induction on the derivation of Λ-reduction of λ^{MD}. We show only the main case.

Case $(\Lambda\alpha.M)\ A \longrightarrow_{\Lambda} M[\alpha \mapsto A]$**:**

By the definition of $\natural$, $\natural((\Lambda\alpha.M)\ A) = \natural(M)$. Because $\natural(M)$ does not contain α, $\natural(M[\alpha \mapsto A]) = \natural(M)$. □

Lemma 10 (Preservation of $\blacklozenge$-Reduction in $\natural$)**.** *If* $\Gamma \vdash_{\Sigma} M : \tau@A$ *and* $M \longrightarrow_{\blacklozenge} N$ *in* λ^{MD} *then* $\natural(M) = \natural(N)$.

Proof. We prove by induction on the derivation of $\blacklozenge$-reduction of λ^{MD}. We show only the main case.

Case $\blacktriangleleft_{\alpha}\blacktriangleright_{\alpha} M \longrightarrow_{\blacklozenge} M$ **:**

By the definition of $\natural$, $\natural(\blacktriangleleft_{\alpha}\blacktriangleright_{\alpha} M) = \natural(M)$. □

To prove Strong Normalization, we prove that there is no infinite sequence composed of Λ and $\blacklozenge$-reductions, first.

Lemma 11 (Strong Normalization without β-reduction)**.** *If* $\Gamma \vdash_{\Sigma} M_1 : \tau@A$ *then there is no infinite sequence of terms* $\{M_i\}_{i\geq 1}$ *which satisfies* $M_i \longrightarrow_{\Lambda} M_{i+1}$ *or* $M_i \longrightarrow_{\blacklozenge} M_{i+1}$ *for* $i \geq 1$.

Proof. We write the number of Λ in term M as $\#\Lambda_M$ and the number of $\blacktriangleright$ in term M as $\#\blacktriangleright_M$. For $i \geq 1$, the pair $(\sharp\Lambda_{M_i}, \sharp\blacktriangleright_{M_i})$ is strictly greater than the pair $(\#\Lambda_{M_{i+1}}, \#\blacktriangleright_{M_{i+1}})$ in lexicographical order because Λ-reduction reduces the number of Λ by one and $\blacklozenge$-reduction reduces the number of $\blacktriangleright$ by one preserving the number of Λ. Now, the lemma follows from well-foundedness of pairs of natural number.

□

Then, we can prove Strong Normalization of λ^{MD}.

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

Proof. If there is an infinite reduction sequence in λ^{MD} then there are infinite β-reductions in the sequence. If there are only finite β-reductions in the sequence, we can construct another infinite reduction sequence which is composed only of

Λ-reduction and $\blacklozenge$-reduction. However, it contradicts Lemma 11.

Then, we show Strong Normalization by proof by contradiction.

We assume that there is an infinite reduction sequence $\{M_i\}$ of λ^{MD} and $\Gamma \vdash_\Sigma M_1 : \tau@A$. Thanks to discussion above, we can assume it contains infinite β-reductions. From $\{M_i\}$, we can construct another infinite reduction sequence $\{\natural(M_i)\}$. From Lemma 8, 9, 10, there are infinite β-reductions in $\{\natural(M_i)\}$.

However, $\natural(\Gamma) \vdash_{\natural(\Sigma)} \natural(M_1) : \natural(\tau)$ from Lemma 7 therefore the existence of $\{\natural(M_i)\}$ contradicts Strong Normalization of LF. □

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 [17] 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.

Lemma 12 (Staged Reduction and Normal Reduction)**.** *If $M \longrightarrow_s M'$, then $M \longrightarrow M'$.*

Proof. Easy from the definition of $M \longrightarrow_s M'$. □

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 4 (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. By straightforward induction on typing derivations. □

The type environment Γ of a statement usually must be empty; in other words, the term must 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.

Theorem 5 (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* $M \longrightarrow_s M'$*.*

4.2 Properties of Algorithmic Typing

In this subsection, we prove properties of algorithmic typing of λ^{MD}. The main purpose of this subsection is prove the completeness and soundness of algorithmic typing which gives the equivalence with the original typing. We start from proving properties of algorithmic reduction because it is used to define term equivalence in QA-ANF. ANF is a function which returns normal form of a term in algorithmic reduction and $\equiv_\alpha$ means α equivalence.

$$\frac{\mathrm{ANF}(M) \equiv_\alpha \mathrm{ANF}(N)}{\Gamma \vdash\!\blacktriangleright M \equiv N@A}\ (\text{QA-ANF})$$

First, we prove the uniqueness of $\mathrm{ANF}(M)$ for all typed term M using Strong Normalization and Confluence of algorithmic reduction. Fortunately, we can prove these two properties similarly to the original reduction.

Lemma 13 (Strong Normalization of Algorithmic Reduction)**.** *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau@A$*, there is no infinite sequence of* $\longrightarrow_A$ *reductions from* M*.*

Proof. Proved by using $\natural$ function. □

Lemma 14 (Confluence of Algorithmic Reduction)**.** *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau@A$*,* $M \longrightarrow_A^* M'$ *and* $M \longrightarrow_A^* M''$ *then there is a* M''' *such that* $M' \longrightarrow_A^* M'''$ *and* $M'' \longrightarrow_A^* M'''$*.*

Proof. By argument similar that in Thorem 3 □

Lemma 15 (Uniqueness of Algorithmic Reduction Normal Form)**.** *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau@A$ *then there is the unique normal form of* M*,* $ANF(M)$*.*

Proof. Immediate from Lemma 13 and 14. □

Then, we prove the relationship between one step algorithmic reduction and substitutions.

Lemma 16 (Algorithmic Reduction and Term Substitution)**.** *If* $\Gamma, z : \xi@B, \Delta \vdash\!\blacktriangleright_\Sigma M : \tau@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma P : \xi@B$ *and* $M \longrightarrow_A M'$ *then* $ANF(M[z \mapsto P]) \equiv_\alpha ANF(M'[z \mapsto P])$.

Proof. By induction on the derivation of $M \longrightarrow_{\mathrm{A}} M'$.

Case A-%**:** $\%M \longrightarrow_{\mathrm{A}} M$ and $\mathrm{FV}(M) = \emptyset$

From $\mathrm{FV}(M) = \emptyset$, $z \notin \mathrm{FV}(M)$ and $z \notin \mathrm{FV}(M')$. Then $\mathrm{ANF}(M) \equiv_\alpha \mathrm{ANF}(M')$ is clear.

Case A-β**:** $(\lambda x : \tau.M)\ N \longrightarrow_{\mathrm{A}} M[x \mapsto N]$

$$
\begin{aligned}
&\mathrm{ANF}(((\lambda x : \tau.M)\ N)[z \mapsto P]) \\
&= \mathrm{ANF}((\lambda x : \tau.M[z \mapsto P])\ N[z \mapsto P]) \\
&= \mathrm{ANF}(M[z \mapsto P][x \mapsto N[z \mapsto P]]) \qquad \text{(From Lemma 15)} \\
&= \mathrm{ANF}(M[x \mapsto N][z \mapsto P]) \\
&= \mathrm{ANF}(M'[z \mapsto P])
\end{aligned}
$$

□

Lemma 17 (Algorithmic Reduction and Stage Substitution)**.** *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau@A$ *and* $M \longrightarrow_A M'$ *then* $ANF(M[\alpha \mapsto B]) \equiv_\alpha ANF(M'[\alpha \mapsto B])$.

Proof. By induction on the derivation of $M \longrightarrow_{\mathrm{A}} M'$. □

Next, we prove that we can derive the same term in two different ways. On the one hand, we may calculate the algorithmic normal form after the substitution. On the other hand, we may calculate the algorithmic normal form after the substitution for the algorithmic normal form. This property (state formally in Lemma 18) is crucial for Term Substitution Lemma of Algorithmic Judgement and Stage Substitution Lemma of Algorithmic Judgement (Lemma 20 and Lemma 21). In [11], they use weak head normal form, but here we use normal form because we couldn't prove this lemma for weak head normal form.

Lemma 18 (Algorithmic Normal Form and Term Substitution)**.** *If* $\Gamma, z : \xi@B, \Delta \vdash\!\blacktriangleright_\Sigma M : \tau@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma P : \xi@B$ *then* $ANF(M[z \mapsto P]) \equiv_\alpha ANF(ANF(M)[z \mapsto P])$.

Proof. Prove by induction on the length n of reduction from M to $\mathrm{ANF}(M)$.

Case $n = 0$**:** $M = \mathrm{ANF}(M)$

It is immediate.

Case $n = 1$**:** $M \longrightarrow_{\mathrm{A}} M'$ and $\mathrm{ANF}(M) = M'$

From Lemma 16.

Case $n > 1$**:** $M \longrightarrow_{\mathrm{A}}^{*} M'$ and $M' \longrightarrow_{\mathrm{A}}^{*} M''$ and $\mathrm{ANF}(M) = M''$

From the induction hypothesis, $\mathrm{ANF}(M[z \mapsto P]) \equiv_\alpha \mathrm{ANF}(M'[z \mapsto P])$. From Lemma 16, $\mathrm{ANF}(M'[z \mapsto P]) \equiv_\alpha \mathrm{ANF}(M''[z \mapsto P])$. Then from the transitivity of $\equiv_\alpha$, $\mathrm{ANF}(M[z \mapsto P]) \equiv_\alpha \mathrm{ANF}(\mathrm{ANF}(M)[z \mapsto P])$. □

Lemma 19 (Algorithmic Normal Form and Stage Substitution)**.** *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau@A$ *then* $ANF(M[\alpha \mapsto B]) \equiv_\alpha ANF(ANF(M)[\alpha \mapsto A])$

Proof. Same as Lemma 18 using Lemma 17 instead of Lemma 16. □

Thanks to Lemma 18 and Lemma 19, we can prove two substitution lemmas.

Lemma 20 (Term Substitution Lemma of Algorithmic Judgement)**.** *If* $\Gamma, z : \xi@B, \Delta \vdash\!\blacktriangleright_\Sigma \mathcal{J}$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma P : \xi@B$ *then* $\Gamma, \Delta[z \mapsto P] \vdash\!\blacktriangleright_\Sigma \mathcal{J}[z \mapsto P]$.

Proof. By induction on the derivation of $\Gamma, z : \xi@B, \Delta \vdash\!\blacktriangleright_\Sigma \mathcal{J}$. Most cases are the same with Lemma 1. We show only different cases.

Case QA-ANF**:** $\Gamma, z : \xi@B, \Delta \vdash\!\blacktriangleright_\Sigma M \equiv N@A$ is derived from $\mathrm{ANF}(M) \equiv_\alpha \mathrm{ANF}(N)$

$\mathrm{ANF}(\mathrm{ANF}(M)[z \mapsto P]) \equiv_\alpha \mathrm{ANF}(\mathrm{ANF}(N)[z \mapsto P])$ is obvious from $\mathrm{ANF}(M) \equiv_\alpha \mathrm{ANF}(N)$. From Lemma 18, $\mathrm{ANF}(M[z \mapsto P]) \equiv_\alpha \mathrm{ANF}(N[z \mapsto P])$. Then $\Gamma, \Delta[z \mapsto P] \vdash\!\blacktriangleright_\Sigma M[z \mapsto P] \equiv N[z \mapsto P]@A$. □

Lemma 21 (Stage Substitution Lemma of Algorithmic Judgement)**.** *If* $\Gamma \vdash\!\blacktriangleright_\Sigma \mathcal{J}$ *then* $\Gamma[\alpha \mapsto B] \vdash\!\blacktriangleright_\Sigma \mathcal{J}[\alpha \mapsto B]$.

Proof. Same as Lemma 20 using Lemma 19 instead of Lemma 18. □

Lemma 20 and 21 enable to prove type preservation of algorithmic reduction.

Lemma 22 (Type Preservation of Algorithmic Reduction)**.** *If* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $M \longrightarrow_A M'$ *then* $\Gamma \vdash_\Sigma M' : \tau@A$*.*

Proof. Similar to Theorem 1, we prove type and equivalence preservation lemma and use them. The different case is A-%. If $\Gamma \vdash_\Sigma \%_\alpha M : \tau@A$, $\%_\alpha M \longrightarrow_{\mathrm{A}} M$ and $\mathrm{FV}(\%_\alpha M) = \emptyset$, M is a closed term. So $\Gamma \vdash_\Sigma M : \tau@A$. □

Lemma 23 (Algorithmic Type Preservation of Algorithmic Reduction)**.** *If* $\Gamma \vdash \blacktriangleright_\Sigma M : \tau@A$ *and* $M \longrightarrow_A M'$ *then* $\Gamma \vdash \blacktriangleright_\Sigma M' : \tau@A$*.*

Proof. Similar to Theorem 1. In the case of A-β and A-Λ we can prove it by Lemma 20 and 21. □

Lemma 24 (Algorithmic Reduction and Term Equivalence)**.** *If* $\Gamma \vdash_\Sigma M : \tau@A$ *and* $M \longrightarrow_A M'$ *then* $\Gamma \vdash_\Sigma M \equiv M' : \tau@A$*.*

Proof. By case analysis on $M \longrightarrow_{\mathrm{A}} M'$.

Case A-%**:** $\%_\alpha M \longrightarrow_{\mathrm{A}} M$ and $\emptyset = \mathrm{FV}(\%_\alpha M)$

Now, we can assume $\Gamma \vdash_\Sigma \%_\alpha M : \tau@A\alpha$ and $\Gamma \vdash_\Sigma M : \tau@A$. We prove $\Gamma \vdash_\Sigma M : \tau@A\alpha$ by replacing all appearances of A with $A\alpha$ in the derivation of $\Gamma \vdash_\Sigma M : \tau@A$. Then, we can prove $\Gamma \vdash_\Sigma \%_\alpha M \equiv M@A\alpha$ using T-Csp. □

Now, we can say term M and $\mathrm{ANF}(M)$ are equivalent.

Lemma 25 (ANF and Term Equivalence)**.** *If* $\Gamma \vdash_\Sigma M : \tau@A$ *then* $\Gamma \vdash_\Sigma M \equiv ANF(M) : \tau@A$*.*

Proof. Corollary of Lemma 22 and 24. □

Finally, we prove the soundness of algorithmic typing. It means that if a term is typed in algorithmic typing then the term is also typed in the original typing. Because algorithmic typing is defined mutually involving other algorithmic judgements, we use simultaneous induction for proof.

Theorem 6 (Soundness of Algorithmic Typing)**.** *If a term is typed in algorithmic rules then the term is typed in normal rules, too.*

- *If* $\Gamma \vdash \blacktriangleright_\Sigma K \ \mathrm{kind}\,@A$ *then* $\Gamma \vdash_\Sigma K \ \mathrm{kind}\,@A$*.*
- *If* $\Gamma \vdash \blacktriangleright_\Sigma \tau :: K@A$ *then* $\Gamma \vdash_\Sigma \tau :: K@A$*.*

- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau@A$ *then* $\Gamma \vdash_\Sigma M : \tau@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma K, K'@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K'@A$ *then* $\Gamma \vdash_\Sigma K \equiv K'@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau, \tau' :: K@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$ *then* $\Gamma \vdash_\Sigma \tau \equiv \tau' :: K@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M, M' :: \tau@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma M \equiv M'@A$ *then* $\Gamma \vdash_\Sigma M \equiv M' :: \tau@A$.

Proof. We can prove using induction on the derivation tree.

Case KA-App**:** $\Gamma \vdash\!\blacktriangleright_\Sigma \sigma\ M :: K[x \mapsto M]@A$
is derived from $\Gamma \vdash\!\blacktriangleright_\Sigma \sigma :: (\Pi x : \tau.K)@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau'@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma \tau' :: *@A$ and $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$.

Considering the derivation of $\Gamma \vdash\!\blacktriangleright_\Sigma \sigma :: (\Pi x : \tau.K)@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma \tau :: *$. Using the induction hypothesis on $\Gamma \vdash_\Sigma \tau, \tau' :: *@A$ and $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$, we get $\Gamma \vdash_\Sigma \tau \equiv \tau' :: *@A$.

From the induction hypothesis, $\Gamma \vdash_\Sigma \sigma :: (\Pi x : \tau.K)@A$ and $\Gamma \vdash_\Sigma M : \tau'@A$. T-Conv and K-App guide to $\Gamma \vdash_\Sigma \sigma\ M :: K[x \mapsto M]@A$.

Case TA-App**:** $\Gamma \vdash\!\blacktriangleright_\Sigma M\ N : \tau[x \mapsto N]@A$
is derived from $\Gamma \vdash\!\blacktriangleright_\Sigma M : (\Pi(x : \sigma).\tau)$, $\Gamma \vdash\!\blacktriangleright_\Sigma N : \sigma'@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma \sigma' :: *@A$ and $\Gamma \vdash\!\blacktriangleright_\Sigma \sigma \equiv \sigma'@A$

Just same as K-App case, $\Gamma \vdash_\Sigma \sigma \equiv \sigma' :: *@A$. From the induction hypothesis, $\Gamma \vdash_\Sigma M : (\Pi(x : \sigma).\tau)@A$ and $\Gamma \vdash_\Sigma N : \sigma'@A$. Then, $\Gamma \vdash_\Sigma M\ N : \tau[x \mapsto N]@A$ from T-Conv and T-App.

Case QA-ANF**:** $\Gamma \vdash\!\blacktriangleright M \equiv N@A$
is derived from $\Gamma \vdash\!\blacktriangleright \mathrm{ANF}(M) \equiv_\alpha \mathrm{ANF}(N)@A$.

Apply the induction hypothesis on $\Gamma \vdash\!\blacktriangleright_\Sigma M, N : \tau@A$, we get $\Gamma \vdash_\Sigma M, N : \tau@A$. Apply Lemma 22 to $\Gamma \vdash_\Sigma M, N : \tau@A$, $\Gamma \vdash_\Sigma \mathrm{ANF}(M), \mathrm{ANF}(N) : \tau@A$. Apply Corollary 25 to $\Gamma \vdash_\Sigma \mathrm{ANF}(M), \mathrm{ANF}(N) : \tau@A$, $\Gamma \vdash_\Sigma M \equiv \mathrm{ANF}(M) : \tau@A$ and $\Gamma \vdash_\Sigma N \equiv \mathrm{ANF}(N) : \tau@A$.

Apply Lemma 23 to $\Gamma \vdash_\Sigma M, N : \tau@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma \mathrm{ANF}(M), \mathrm{ANF}(N) : \tau@A$. $\Gamma \vdash_\Sigma \mathrm{ANF}(M) \equiv \mathrm{ANF}(N) : \tau@A$ from using the induction hypothesis on $\Gamma \vdash\!\blacktriangleright_\Sigma \mathrm{ANF}(M), \mathrm{ANF}(N) : \tau@A$ and $\Gamma \vdash\!\blacktriangleright \mathrm{ANF}(M) \equiv_\alpha \mathrm{ANF}(N)@A$.

Apply Q-Trans to $\Gamma \vdash_\Sigma \mathrm{ANF}(M) \equiv \mathrm{ANF}(N) : \tau@A$ and
$\Gamma \vdash_\Sigma M \equiv \mathrm{ANF}(M) : \tau@A$ and $\Gamma \vdash_\Sigma N \equiv \mathrm{ANF}(N) : \tau@A$, $\Gamma \vdash_\Sigma M \equiv N : \tau@A$

□

Next theorem is the completeness of algorithmic typing. First, we prove symmetricity, reflexivity and transitivity of algorithmic typing which are indispensable for the proof.

Lemma 26 (Symmetricity in Algorithmic Equivalence)**.** *The algorithmic equivalence relationship is symmetrical.*

- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K'@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma K' \equiv K@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau' \equiv \tau@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M \equiv M'@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma M' \equiv M@A$.

Proof. It is obvious because all algorithmic equivalence rules are symmetrical. □

Lemma 27 (Reflexivity in Algorithmic Equivalence)**.** *The algorithmic equivalence relationship is reflexive.*

- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M \equiv M@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma M \equiv M@A$.

Proof. Prove by induction. □

Lemma 28 (Transition in Algorithmic Equivalence)**.** *The algorithmic equivalence relationship is transitive.*

- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K'@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma K' \equiv K''@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K''@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau' \equiv \tau''@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau''@A$.
- *If* $\Gamma \vdash\!\blacktriangleright_\Sigma M \equiv M'@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma M' \equiv M''@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma M \equiv M''@A$.

Proof. Prove by the induction on derivations. In this lemma, there are two hypothesis $\mathcal{J}_1$ and $\mathcal{J}_2$. The last rules of $\mathcal{J}_1$ and $\mathcal{J}_2$ are the same. Because these are algorithmic derivation so the last rule of derivation is decided uniquely by the shape of judgments. Then we can prove this theorem easily from the induction hypothesis and the last rule. □

This lemma is used in Q-% case in the proof of the completeness.

Lemma 29 (Free Variable and Stage)**.** *If* $\Gamma \vdash_\Sigma M : \tau@A\alpha$ *and* $\Gamma \vdash_\Sigma M : \tau@A$

then $FV(M) = \emptyset$.

Proof. Prove by contradiction. Assume $\Gamma \vdash_\Sigma M : \tau@A\alpha$ and $\Gamma \vdash_\Sigma M : \tau@A$ and $\mathrm{FV}(M) \neq \emptyset$. Then, $x : \tau@B, x : \tau@B\alpha \in \Gamma$. However, from well-formedness of Γ, there is no such x. □

This is the last theorem – Completeness. It means that when a term is typed in the original typing, it is also typed in the algorithmic typing.

Theorem 7 (Completeness of Algorithmic Typing)**.**

- *If* $\Gamma \vdash_\Sigma K$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma K$.
- *If* $\Gamma \vdash_\Sigma \tau :: K@A$ *then there is* K' *such that* $\Gamma \vdash\!\blacktriangleright_\Sigma K'@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K'@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau :: K'@A$.
- *If* $\Gamma \vdash_\Sigma M : \tau@A$ *then there is* τ' *such that* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau' :: *@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$ *and* $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau'@A$.
- *If* $\Gamma \vdash_\Sigma K \equiv K'@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K'@A$.
- *If* $\Gamma \vdash_\Sigma \tau \equiv \tau' :: K@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$.
- *If* $\Gamma \vdash_\Sigma M \equiv M' : \tau@A$ *then* $\Gamma \vdash\!\blacktriangleright_\Sigma M \equiv M'@A$.

Proof. Prove by induction on the derivation.

Case K-App**:** $\Gamma \vdash_\Sigma \sigma\ M :: K[x \mapsto M]@A$
is derived from $\Gamma \vdash_\Sigma \sigma :: (\Pi x : \tau.K)@A$ and $\Gamma \vdash_\Sigma M : \tau@A$.

Apply the induction hypothesis on $\Gamma \vdash_\Sigma \sigma :: (\Pi x : \tau.K)@A$, there is a kind J such that $\Gamma \vdash\!\blacktriangleright_\Sigma J@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma \sigma :: J$ and $\Gamma \vdash\!\blacktriangleright_\Sigma J \equiv (\Pi x : \tau.K)@A$. The last rule of the derivation of $\Gamma \vdash\!\blacktriangleright_\Sigma J \equiv (\Pi x : \tau.K)@A$ is QTA-Abs, so $J = \Pi x : \tau'.K'$, $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$ and $\Gamma, x : \tau \vdash\!\blacktriangleright_\Sigma K \equiv K'@A$. Use Lemma 3 and the induction hypothesis on $\Gamma \vdash_\Sigma M : \tau@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma \tau :: *@A$. Now, we can apply KA-App to $\Gamma \vdash\!\blacktriangleright_\Sigma \sigma :: \Pi x : \tau'.K'@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma M : \tau@A$ and $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$ and get $\Gamma \vdash\!\blacktriangleright_\Sigma \sigma\ M :: K'[x \mapsto M]@A$. From Lemma 20, $\Gamma \vdash\!\blacktriangleright_\Sigma K[x \mapsto M] \equiv K'[x \mapsto M]@A$.

Case K-Conv**:** $\Gamma \vdash_\Sigma \tau :: J@A$
is derived from $\Gamma \vdash_\Sigma \tau :: K@A$ and $\Gamma \vdash_\Sigma K \equiv J@A$

Apply the induction hypothesis on $\Gamma \vdash_\Sigma \tau :: K@A$, there is a kind K' such that $\Gamma \vdash\!\blacktriangleright_\Sigma K'@A$ and $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv K'@A$ and $\Gamma \vdash\!\blacktriangleright_\Sigma \tau :: K'@A$. Apply the induction

hypothesis on $\Gamma \vdash_\Sigma K \equiv J@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma K \equiv J@A$. From Lemma 26 and 28, $\Gamma \vdash\!\blacktriangleright_\Sigma J \equiv K'@A$.

Case QT-App**:** $\Gamma \vdash_\Sigma \tau\ M \equiv \sigma\ N :: K[x \mapsto M]@A$
is derived from $\Gamma \vdash_\Sigma \tau \equiv \sigma :: (\Pi x : \rho.K)@A$ and
$\Gamma \vdash_\Sigma M \equiv N : \rho@A$

From the induction hypothesis, $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \sigma@A$ and $\Gamma \vdash\!\blacktriangleright_\Sigma M \equiv N@A$. Use TA-App, $\Gamma \vdash\!\blacktriangleright_\Sigma \tau\ M \equiv \sigma\ N$.

Case T-Conv**:** $\Gamma \vdash_\Sigma M : \tau'@A$
is derived from $\Gamma \vdash_\Sigma M : \tau@A$ and $\Gamma \vdash_\Sigma \tau \equiv \tau' :: K@A$

Apply the induction hypothesis on $\Gamma \vdash_\Sigma \tau \equiv \tau' :: K@A$, $\Gamma \vdash\!\blacktriangleright_\Sigma \tau \equiv \tau'@A$. Use Lemma 3 and the induction hypothesis to $\Gamma \vdash_\Sigma M : \tau@A$, we get $\Gamma \vdash\!\blacktriangleright_\Sigma \tau :: *@A$. Now we can see τ as τ' in the second item of Theorem 7.

Case Q-%**:** $\Gamma \vdash_\Sigma \%_\alpha M \equiv M : \tau@A\alpha$ is derived from
$\Gamma \vdash_\Sigma M : \tau@A\alpha$ and $\Gamma \vdash_\Sigma M : \tau@A$

From Lemma 29, $\mathrm{FV}(M) = \emptyset$. Then, $\%_\alpha M \longrightarrow_{\mathrm{A}} M$ and $\mathrm{ANF}(\%_\alpha M) = \mathrm{ANF}(M)$. So, $\Gamma \vdash\!\blacktriangleright_\Sigma \%_\alpha M \equiv M$ by QA-ANF.

Case QK-Sym**:**

It is obvious from Lemma 26.

Case QK-Refl**:**

It is obvious from Lemma 27.

Case QK-Trans**:**

It is obvious from Lemma 28. □

From Theorem 6 and 7, we can say the original typing and the algorithmic typing are equivalent.

Chapter 5 Related Work

MetaOCaml is a programming language with quoting, unquoting, run, and CSP. Kiselyov [1] 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 [18] and Davies [19], who found Curry-Howard correspondence between multi-stage calculi and modal logic. In particular, Davies' λ° [19] has been a basis for several multi-stage calculi with quasi-quotation. λ° did not have operators for run and CSP; a few studies [20, 21] 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 [6]. Finally, Taha and Nielsen invented the concept of environment classifiers [7] 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 [22] and Nanevski and Pfenning [23].

Later, Tsukada and Igarashi [14] 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 [10] 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 [24]. Among many dependent type systems such as λ^Π [25], The Calculus of Constructions [26], and Edinburgh LF [15], we base our work on λLF [11] (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 [9] and typed abstract syntax trees [27, 28]. The vector addition discussed in Section Chapter 3 is also such an example.

The use of dependent types for code generation is studied by Chlipala [29] and Ebner et al. [30]. 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 [12] designed $\lambda_{H\circ}$ by introducing the concept of stage into an existing dependent type system λ_H [31]. However, $\lambda_{H\circ}$ is equipped with neither run nor CSP. Forgarty, Pasalic, Siek and Taha [32] extended the type system of MetaOCaml with indexed types. With this extension, types can be indexed with a Coq term. Chen and Xi [33] 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 [13] 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. Gratzer et al. introduced modalities in dependent type thoory in [34]. It will helps us to find the logic which corresponds to λ^{MD}.

Berger and Tratt [35] gave program logic for Mini-ML$_e^{\Box}$ [18], which would allow fine-grained reasoning about the behavior of code generators. However, it cannot manipulate open code which ours can deal with.

Chapter 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. Because the generated code by multi-stage programming is sometimes specialized in given inputs, it cannot be used with specific inputs. This expansion can assure it is only used with desired inputs.

Technically, 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. Furthermore, we design algorithmic typing rules for λ^{MD}, which we need to implement a type inference program and prove it is equivalent to original typing rules.

The main future work is investigating the Curry-Howard isomorphism to find a logic corresponding to λ^{MD}. Tsukada and Igarashi [14] found that a kind of modal logic corresponds to stage, and Gratzer et al. [34] investigated a dependent type theory with modalities. However, they have not explored the CSP operator, which is the critical point of our research.

Acknowledgments

First of all, I thank to my supervisor, Prof. Atsushi Igarashi. Without his support, I cannot finish my thesis. He is the most powerful and smart person who I have met.

Second, I thank to all members in the laboratory. They made my life in my master cource fruitful and pleasant.

At the last, I am grateful to all restaurants around the laboratory, Cofe Collection, Matsunosuke, Kitchen Gorilla, and Mikaen. These restaurants encourage me to proceed my research.

References

[1] Kiselyov, O.: *Reconciling Abstraction with High Performance: A MetaOCaml approach*, NOW Publishing (2018).

[2] Mainland, G.: Explicitly Heterogeneous Metaprogramming with MetaHaskell, *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, New York, NY, USA, ACM, pp. 311–322 (2012).

[3] Taha, W.: A Gentle Introduction to Multi-stage Programming, Part II, *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).

[4] Calcagno, C., Taha, W., Huang, L. and Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection, *International Conference on Generative Programming and Component Engineering*, Springer, pp. 57–76 (2003).

[5] Kiselyov, O.: The Design and Implementation of BER MetaOCaml - System Description, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pp. 86–102 (2014).

[6] Taha, W. and Sheard, T.: MetaML and Multi-stage Programming with Explicit Annotations, *Theor. Comput. Sci.*, Vol. 248, No. 1-2, pp. 211–242 (2000).

[7] Taha, W. and Nielsen, M. F.: Environment Classifiers, *Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, New York, NY, USA, ACM, pp. 26–37 (2003).

[8] Milner, R.: A Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences*, Vol. 17, pp. 348–375 (1978).

[9] Xi, H. and Pfenning, F.: Eliminating Array Bound Checking Through Dependent Types, *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pp. 249–257 (1998).

[10] Hanada, Y. and Igarashi, A.: On Cross-Stage Persistence in Multi-Stage

Programming, *Functional and Logic Programming* (Codish, M. and Sumii, E.(eds.)), Cham, Springer International Publishing, pp. 103–118 (2014).

[11] Aspinall, D. and Hofmann, M.: Dependent Types, *Advanced topics in types and programming languages* (Pierce, B. C.(ed.)), MIT press, chapter 2 (2005).

[12] Pasalic, E., Taha, W. and Sheard, T.: Tagless Staged Interpreters for Typed Languages, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, New York, NY, USA, ACM, pp. 218–229 (2002).

[13] Brady, E. and Hammond, K.: Dependently Typed Meta-programming, *Trends in Functional Programming* (2006).

[14] Tsukada, T. and Igarashi, A.: A Logical Foundation for Environment Classifiers, *Logical Methods in Computer Science*, Vol. 6, No. 4 (2010).

[15] Harper, R., Honsell, F. and Plotkin, G.: A framework for defining logics, *Journal of the ACM (JACM)*, Vol. 40, No. 1, pp. 143–184 (1993).

[16] Pierce, B. C.: *Types and Programming Languages*, MIT Press (2002).

[17] Baader, F. and Nipkow, T.: *Term rewriting and all that*, Cambridge University Press (1998).

[18] Davies, R. and Pfenning, F.: A Modal Analysis of Staged Computation, *Journal of the ACM*, Vol. 48, No. 3, pp. 555–604 (2001).

[19] Davies, R.: A temporal-logic approach to binding-time analysis, *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, IEEE, pp. 184–195 (1996).

[20] Benaissa, Z. E.-A., Moggi, E., Taha, W. and Sheard, T.: Logical modalities and multi-stage programming, *Federated logic conference (FLoC) satellite workshop on intuitionistic modal logics and applications (IMLA)* (1999).

[21] Moggi, E., Taha, W., Benaissa, Z. E.-A. and Sheard, T.: An Idealized MetaML: Simpler, and More Expressive, *Proc. of European Symposium on Programming (ESOP'99)*, Lecture Notes in Computer Science, Vol. 1576, pp. 193–207 (1999).

[22] Kim, I., Yi, K. and Calcagno, C.: A polymorphic modal type system for lisp-like multi-staged languages, *Proc. of ACM SIGPLAN-SIGACT Sym-*

posium on Principles of Programming Languages (POPL 2003), pp. 257–268 (2006).

[23] Nanevski, A. and Pfenning, F.: Staged computation with names and necessity, *J. Funct. Program.*, Vol. 15, No. 5, pp. 893–939 (2005).

[24] Martin-Löf, P.: An Intuitionistic Theory of Types: Predicative Part, *Logic Colloquium '73*, Vol. 80 (1973).

[25] Meyer, A. R. and Reinhold, M. B.: "Type" is Not a Type, *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, New York, NY, USA, ACM, pp. 287–295 (1986).

[26] Coquand, T. and Huet, G.: The Calculus of Constructions, *Inf. Comput.*, Vol. 76, No. 2-3, pp. 95–120 (1988).

[27] Leijen, D. and Meijer, E.: Domain specific embedded compilers, *Proc. of the 2nd Conference on Domain-Specific Languages (DSL '99)*, pp. 109–122 (1999).

[28] Xi, H., Chen, C. and Chen, G.: Guarded recursive datatype constructors, *Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pp. 224–235 (2003).

[29] Chlipala, A.: Ur: Statically-typed Metaprogramming with Type-level Record Computation, *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, New York, NY, USA, ACM, pp. 122–133 (2010).

[30] Ebner, G., Ullrich, S., Roesch, J., Avigad, J. and de Moura, L.: A metaprogramming framework for formal verification, *PACMPL*, Vol. 1, No. ICFP, pp. 34:1–34:29 (2017).

[31] Shao, Z., Saha, B., Trifonov, V. and Papaspyrou, N.: A Type System for Certified Binaries, *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, New York, NY, USA, ACM, pp. 217–232 (2002).

[32] Fogarty, S., Pasalic, E., Siek, J. and Taha, W.: Concoqtion: Indexed Types Now!, *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, New

York, NY, USA, ACM, pp. 112–121 (2007).

[33] Chen, C. and Xi, H.: Meta-programming Through Typeful Code Representation, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, New York, NY, USA, ACM, pp. 275–286 (2003).

[34] Gratzer, D., Sterling, J. and Birkedal, L.: Implementing a Modal Dependent Type Theory, *Proc. ACM Program. Lang.*, Vol. 3, No. ICFP (2019).

[35] Berger, M. and Tratt, L.: Program Logics for Homogeneous Generative Run-Time Meta-Programming, *Logical Methods in Computer Science*, Vol. Volume 11, Issue 1 (2015).

Appendix

A.1 Full Definition of $\lambda^{\mathbf{MD}}$

A.1.1 Syntax

$$
\begin{array}{rl}
\text{Terms} & M, N, L, O, P ::= c \mid x \mid \lambda x : \tau.M \mid M\ M \mid \blacktriangleright_\alpha M \\
& \qquad \mid \blacktriangleleft_\alpha M \mid \Lambda\alpha.M \mid M\ A \mid \%_\alpha M \\
\text{Types} & \tau, \sigma, \rho, \pi, \xi ::= X \mid \Pi x : \tau.\tau \mid \tau\ M \mid \triangleright_\alpha \tau \mid \forall \alpha.\tau \\
\text{Kinds} & K, J, I, H, G ::= * \mid \Pi x : \tau.K \\
\text{Type environments} & \Gamma ::= \emptyset \mid \Gamma, x : \tau @ A \\
\text{Signature} & \Sigma ::= \emptyset \mid \Sigma, X :: K \mid \Sigma, c : \tau \\
\text{Stage variables} & \alpha, \beta, \gamma, ... \\
\text{Stage} & A, B, C, ... \\
\text{Variables} & x, y, z, ... \\
\text{Type variables} & X, Y, Z, ...
\end{array}
$$

A.1.2 Reduction

$\boxed{M \longrightarrow M'}$ **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.

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

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

A.1.3 Type System

$\boxed{\vdash \Sigma}$ **Well-formed signatures**

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

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

$\boxed{\vdash_\Sigma \Gamma}$ **Well-formed type environments**

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

$\boxed{\Gamma \vdash K \ \text{kind}\,@A}$ **Well-formed kinds**

$$\frac{}{\Gamma \vdash_\Sigma * \ \text{kind}\,@A}\ (\text{W-Star})$$

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

$\boxed{\Gamma \vdash_\Sigma \tau :: K@A}$ **Kinding**

$$\frac{X :: K \in \Sigma}{\Gamma \vdash_\Sigma X :: K@A}\ (\text{K-TConst})$$

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

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

$$\frac{\Gamma \vdash_\Sigma \tau :: K@A \qquad \Gamma \vdash_\Sigma K \equiv J@A}{\Gamma \vdash_\Sigma \tau :: J@A}\ (\text{K-Conv}) \qquad \frac{\Gamma \vdash_\Sigma \tau :: *@A\alpha}{\Gamma \vdash_\Sigma \triangleright_\alpha \tau :: *@A}\ (\text{K-}\triangleright)$$

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

$\boxed{\Gamma \vdash_\Sigma M : \tau@A}$ **Typing**

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash_\Sigma M : \tau@A \qquad \Gamma \vdash_\Sigma \tau :: *@A\alpha}{\Gamma \vdash_\Sigma \%_\alpha M : \tau@A\alpha}\ (\text{T-Csp})$$

$\boxed{\Gamma \vdash_\Sigma K \equiv J@A}$ **Kind Equivalence**

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

$$\frac{\Gamma \vdash_\Sigma K\ \ \mathrm{kind}\,@A}{\Gamma \vdash_\Sigma K \equiv K@A}\ (\text{QK-Refl}) \qquad \frac{\Gamma \vdash_\Sigma K \equiv J@A}{\Gamma \vdash_\Sigma J \equiv K@A}\ (\text{QK-Sym})$$

$$\frac{\Gamma \vdash_\Sigma K \equiv J@A \qquad \Gamma \vdash_\Sigma J \equiv I@A}{\Gamma \vdash_\Sigma K \equiv I@A}\ (\text{QK-Trans})$$

$\boxed{\Gamma \vdash_\Sigma S \equiv T :: K@A}$ **Type Equivalence**

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

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

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

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

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

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

$\boxed{\Gamma \vdash_\Sigma M \equiv N : \tau@A}$ **Term Equivalence**

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

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

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

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

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

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

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

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

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

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

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

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

A.1.4 Algorithmic Typing of $\lambda^{\mathbf{MD}}$

$\boxed{\vdash\blacktriangleright\Sigma}$ **Well-formed signatures**

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

$$\frac{\vdash\blacktriangleright\Sigma \qquad \vdash\blacktriangleright\tau :: *@\varepsilon \qquad c \notin dom(\Sigma)}{\vdash\blacktriangleright\Sigma, c : \tau}$$

$\boxed{\vdash\blacktriangleright_\Sigma\Gamma}$ **Well-formed type environments**

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

$\boxed{\Gamma \vdash\blacktriangleright_\Sigma K\ \ \mathrm{kind}\,@A}$ **Well-formed kinds**

$$\frac{}{\Gamma \vdash\blacktriangleright_\Sigma *\ \ \mathrm{kind}\,@A}\ (\text{W-Star})$$

$$\frac{\Gamma \vdash\blacktriangleright_\Sigma\tau :: *@A \qquad \Gamma, x : \tau@A \vdash\blacktriangleright_\Sigma K\ \ \mathrm{kind}\,@A}{\Gamma \vdash\blacktriangleright_\Sigma(\Pi x : \tau.K)\ \ \mathrm{kind}\,@A}\ (\text{W-Abs})$$

$\boxed{\Gamma \vdash\blacktriangleright_\Sigma\tau :: K@A}$ **Kinding**

$$\frac{X :: K \in \Sigma}{\Gamma \vdash\blacktriangleright_\Sigma X :: K@A}\ (\text{KA-TConst})$$

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

$$\frac{\begin{array}{c}\Gamma \vdash\blacktriangleright_\Sigma\sigma :: (\Pi x : \tau.K)@A \qquad \Gamma \vdash\blacktriangleright_\Sigma M : \tau'@A \\ \Gamma \vdash\blacktriangleright_\Sigma\tau' :: *@A \qquad \Gamma \vdash\blacktriangleright_\Sigma\tau \equiv \tau'@A\end{array}}{\Gamma \vdash\blacktriangleright_\Sigma\sigma\ M :: K[x \mapsto M]@A}\ (\text{KA-App})$$

$$\frac{\Gamma \vdash\blacktriangleright_\Sigma\tau :: *@A\alpha}{\Gamma \vdash\blacktriangleright_\Sigma\triangleright_\alpha\tau :: *@A}\ (\text{KA-}\triangleright)$$

$$\frac{\Gamma \vdash\blacktriangleright_\Sigma\tau :: K@A \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\mathrm{A})}{\Gamma \vdash\blacktriangleright_\Sigma\forall\alpha.\tau :: K@A}\ (\text{KA-Gen})$$

$\boxed{\Gamma \vdash\blacktriangleright_\Sigma M : \tau@A}$ **Algorithmic Typing**

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

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

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} M : (\Pi(x:\sigma).\tau) \qquad \Gamma \vdash^{\blacktriangleright}_{\Sigma} N : \sigma'@A \qquad \Gamma \vdash^{\blacktriangleright}_{\Sigma} \sigma' :: *@A \qquad \Gamma \vdash^{\blacktriangleright}_{\Sigma} \sigma \equiv \sigma'@A}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} M\ N : \tau[x \mapsto N]@A}\ (\text{TA-App})$$

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} M : \tau@A\alpha}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \blacktriangleright_\alpha M : \triangleright_\alpha \tau@A}\ (\text{TA-}\blacktriangleright) \qquad \frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} M : \triangleright_\alpha \tau@A}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \blacktriangleleft_\alpha M : \tau@A\alpha}\ (\text{TA-}\blacktriangleleft)$$

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} M : \tau@A \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\mathrm{A})}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \Lambda\alpha.M : \forall\alpha.\tau@A}\ (\text{TA-Gen})$$

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} M : \forall\alpha.\tau@A}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} M\ A : \tau[\alpha \mapsto A]@A}\ (\text{TA-Ins})$$

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} M : \tau@A \qquad \Gamma \vdash^{\blacktriangleright}_{\Sigma} \tau :: *@A\alpha}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \%_\alpha M : \tau@A\alpha}\ (\text{TA-Csp})$$

$\boxed{\Gamma \vdash^{\blacktriangleright}_{\Sigma} K \equiv J@A}$ **Algorithmic Kind Equivalence**

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \tau \equiv \sigma@A \qquad \Gamma, x:\tau \vdash^{\blacktriangleright}_{\Sigma} K \equiv J@A}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \Pi x:\tau.K \equiv \Pi x:\sigma.J@A}\ (\text{QKA-Abs})$$

$$\frac{}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} * \equiv *@A}\ (\text{QKA-Refl})$$

$\boxed{\Gamma \vdash^{\blacktriangleright}_{\Sigma} S \equiv T@A}$ **Algorithmic Type Equivalence**

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \tau \equiv \sigma@A \qquad \Gamma, x:\tau \vdash^{\blacktriangleright}_{\Sigma} \rho \equiv \pi@A}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \Pi x:\tau.\rho \equiv \Pi x:\sigma.\pi@A}\ (\text{QTA-Abs})$$

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \tau \equiv \sigma@A \qquad \Gamma \vdash^{\blacktriangleright}_{\Sigma} M \equiv N@A}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \tau\ M \equiv \sigma\ N@A}\ (\text{QTA-App})$$

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \tau \equiv \sigma@A\alpha}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \triangleright_\alpha \tau \equiv \triangleright_\alpha \sigma@A}\ (\text{QTA-}\triangleright)$$

$$\frac{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \tau \equiv \sigma@A \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\mathrm{A})}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} \forall\alpha.\tau \equiv \forall\alpha.\sigma@A}\ (\text{QTA-Gen})$$

$$\frac{X :: K \in \Sigma}{\Gamma \vdash^{\blacktriangleright}_{\Sigma} X \equiv X@A}\ (\text{QTA-Const})$$

$\boxed{M \longrightarrow_{\mathrm{A}} M'}$ **Algorithmic Reduction**

$$(\lambda x : \tau.M)N \longrightarrow_{\mathrm{A}} M[x \mapsto N] \qquad \text{(A-}\beta\text{)}$$

$$\blacktriangleleft_\alpha(\blacktriangleright_\alpha M) \longrightarrow_{\mathrm{A}} M \qquad \text{(A-}\blacktriangleleft\blacktriangleright\text{)}$$

$$(\Lambda\alpha.M)\ A \longrightarrow_{\mathrm{A}} M[\alpha \mapsto A] \qquad \text{(A-}\Lambda\text{)}$$

$$\%_\alpha M \longrightarrow_{\mathrm{A}} M \qquad (\emptyset = \mathrm{FV}(\%_\alpha M)) \qquad \text{(A-}\%\text{)}$$

$M \longrightarrow_{\mathrm{A}} M'$ is defined by taking the minimum congruences of these four reductions on terms.

$\boxed{\Gamma \vdash\!\blacktriangleright M \equiv M'@A}$ **Algorithmic Term Equivalence**

$$\frac{\mathrm{ANF}(M) \equiv_\alpha \mathrm{ANF}(N)}{\Gamma \vdash\!\blacktriangleright M \equiv N@A}\ \text{(QA-ANF)}$$