Meta-Programming through Typeful Code Representation*

Chiyan Chen and Hongwei Xi

Computer Science Department
Boston University

{chiyan, hwxi}@cs.bu.edu

Abstract

By allowing the programmer to write code that can generate code at run-time, meta-programming offers a powerful approach to program construction. For instance, meta-programming can often be employed to enhance program efficiency and facilitate the construction of generic programs. However, meta-programming, especially in an untyped setting, is notoriously error-prone. In this paper, we aim at making meta-programming less error-prone by providing a type system to facilitate the construction of correct meta-programs. We first introduce some code constructors for constructing typeful code representation in which program variables are replaced with deBruijn indices, and then formally demonstrate how such typeful code representation can be used to support meta-programming. The main contribution of the paper lies in recognition and then formalization of a novel approach to typed meta-programming that is practical, general and flexible.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative Languages*

General Terms

Languages, Theory

Keywords

Meta-Programming, Multi-Level Staged Programming, Typeful Code Representation

*Partially supported by NSF grants no. CCR-0224244 and no. CCR-0229480

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

1 Introduction

Situations often arise in practice where there is a need for writing programs that can generate programs at run-time. For instance, there are numerous cases (kernel implementation [18], graphics [26], interactive media [10], method dispatch in object-oriented languages [9, 2], etc.) where run-time code generation can be employed to reap significant gain in run-time performance [17]. To illustrate this point, we define a function *evalPoly* as follows in Scheme for evaluating a polynomial p at a given point x.

```
(define (evalPoly p x)
  (if (null? p)
      0
      (+ (car p) (* x (evalPoly (cdr p) x)))))
```

Note that we use a nonempty list $(a_0\ a_1\ \ldots\ a_n)$ in Scheme to represent the polynomial $a_n x^n + \ldots + a_1 x + a_0$. We now define a function *sumPoly* such that $(sumPoly\ \ p\ \ xs)$ returns the sum of the values of a polynomial p at the points listed in xs.

```
(define (sumPoly p xs)
  (if (null? xs)
      0
      (+ (evalPoly p (car xs))
         (sumPoly p (cdr xs)))))
```

When calling *sumPoly*, we generally need to evaluate a *fixed* polynomial *repeatedly* at different points. This suggests that we implement *sumPoly* with the following strategy so as to make *sumPoly* more efficient.

We first define a function *genEvalPoly* as follows, where we make use of the backquote/comma notation in Scheme.

```
(define (genEvalPoly p)
  (define (aux p x)
    (if (null? p)
        0
        `(+ ,(car p)
            (* ,x ,(aux (cdr p) x)))))
  `(lambda (x) ,(aux p `x)))
```

When applied to a polynomial p, *genEvalPoly* returns an s-expression that represents a procedure (in Scheme) for evaluating p. For instance, $(genEvalPoly\ \ '(3\ 2\ 1))$ returns the following s-expression,

$$(lambda\ (x)\ (+\ 3\ (*\ x\ (+\ 2\ (*\ x\ (+\ 1\ (*\ x\ 0)))))))$$

which represents a procedure for evaluating the polynomial $x^2 + 2x + 3$. Therefore, given a polynomial p, we can call $(eval\ (genEvalPoly\ p)\ '())$ to generate a procedure *proc* for

evaluating p[1]; presumably, (*proc x*) should execute faster than (*evalPoly p x*) does. This leads to the following (potentially) more efficient implementation of *sumPoly*.

```
(define (sumPoly p xs)
  (define proc (eval (genEvalPoly p) '()))
  (define (aux xs)
     (if (null? xs)
         0
         (+ (proc (car xs)) (aux xs))))
  (aux xs))
```

Meta-programming, though useful, is notoriously error-prone in general and approaches such as hygenic macros [11] have been proposed to address the issue. Programs generated at run-time often contain type errors or fail to be closed, and errors in meta-programs are generally more difficult to locate and fix than those in (ordinary) programs. This naturally leads to a need for typed meta-programming so that types can be employed to capture errors in meta-programs at compile-time.

The first and foremost issue in typed meta-programming is the need for properly representing typed code. Intuitively, we need a type constructor $(\cdot)$*code* such that for a given type τ, (τ)*code* is the type for expressions representing code of type τ. Also, we need a function *run* such that for a given expression e of type (τ)*code*, *run*(e) executes the code represented by e and then returns a value of type τ when the execution terminates. Note that we cannot in general execute open code, that is, code containing free program variables. Therefore, for each type τ, the type (τ)*code* should only be for expressions representing closed code of type τ.

A common approach to capturing the notion of closed code is through higher-order abstract syntax (h.o.a.s.) [3, 24, 23]. For instance, the following declaration in Standard ML (SML) [19] declares a datatype for representing pure untyped closed λ-expressions:

```
datatype exp =
  Lam of (exp -> exp) | App of exp * exp
```

As an example, the representation of the untyped λ-expression $\lambda x.\lambda y.y(x)$ is given below:

```
Lam(fn (x:exp) => Lam (fn (y:exp) => App(y, x))
```

Although it seems difficult, if not impossible, to declare a datatype in ML for precisely representing typed λ-expressions, this can be readily done if we extend ML with guarded recursive (g.r.) datatype constructors [35]. For instance, we can declare a g.r. datatype constructor $(\cdot)$*HOAS* and associate with it two value constructors *HOASlam* and *HOASapp* that are assigned the following types, respectively.[2]

$$\forall\alpha\forall\beta.((\alpha)HOAS \to (\beta)HOAS) \to (\alpha \to \beta)HOAS$$
$$\forall\alpha\forall\beta.(\alpha \to \beta)HOAS * (\alpha)HOAS \to (\beta)HOAS)$$

Intuitively, for a given type τ, (τ)*HOAS* is the type for h.o.a.s. trees that represent closed code of type τ. As an example, the h.o.a.s. representation of the simply typed λ-expression $\lambda x : int.\lambda y : int \to int.y(x)$ is given below,

```
HOASlam(fn x:int HOAS =>
  HOASlam(fn y:(int -> int) HOAS => HOASapp(y, x)))
```

which has the type $(int \to (int \to int) \to int)HOAS$.

[1]Note that *eval* is a built-in function in Scheme that takes an s-expression and an environment as its arguments and returns the value of the expression represented by the s-expression.

[2]Note that, unlike a similar inductively defined type constructor [25], *HOAS* cannot be inductively defined.

By associating with *HOAS* some extra value constructors, we can represent closed code of type τ as expressions of type (τ)*HOAS*. In other words, we can define $(\cdot)$*code* as $(\cdot)$*HOAS*. The function *run* can then be implemented by first translating h.o.a.s. trees into (untyped) first-order abstract syntax (f.o.a.s.) trees[3] and then compiling the f.o.a.s. trees in a standard manner. Please see a recent paper [35] for more details on such an implementation.

Though clean and elegant, there are some serious problems with representing code as h.o.a.s trees. In general, it seems rather difficult, if not impossible, to manipulate open code in a satisfactory manner when higher-order code representation is chosen. On the other hand, there is often a need to directly handle open code when meta-programs are constructed. For instance, in the definition of the function *genEvalPoly*, the auxiliary function *aux* returns some open code containing one free program variable (which is closed later). We feel it may make meta-programming too restrictive if open code manipulation is completely disallowed.

Furthermore, higher-order code representation may lead to a subtle problem. Suppose we need to convert the following h.o.a.s. tree t, which has the type $((int)HOAS \to int)HOAS$, into some f.o.a.s. tree in order to run the code represented by t:

```
HOASlam (fn (x: (int HOAS) HOAS) => run x)
```

We then need to apply the function *run* to a variable ranging over expressions of type $((int)HOAS)HOAS$ when making this conversion, which unfortunately causes a run-time error. This is precisely the problem of *free variable evaluation*, a.k.a. *open code extrusion*, which we encounter when trying to evaluate the code:

```
<fn x:<int> => ~(run <x>)>
```

in MetaML [33].

In this paper, we choose a form of first-order abstract syntax trees to represent code that not only support direct open code manipulation but also avoid the problem of free variable evaluation. As for the free program variables in open code, we use deBruijn indices [8] to represent them. For instance, we can declare the following datatype in Standard ML to represent pure untyped λ-expressions.

```
datatype exp =
  One | Shi of exp | Lam of exp | App of exp * exp
```

We use *One* for the first free variable in a λ-expression and *Shi* for shifting each free variable in a λ-expression by one index. As an example, the expression $\lambda x.\lambda y.y(x)$ can be represented as follows:

$$Lam(Lam(App(One, Shi(One))))$$

For representing typed expressions, we refine *exp* into types of the form $\langle G, \tau\rangle$, where $\langle\cdot,\cdot\rangle$ is a binary type constructor and G stands for type environments, which are represented as sequences of types; an expression of type $\langle G, \tau\rangle$ represents some code of type τ in which the free variables are assigned types by G, and therefore the type for closed code of type τ is simply $\langle \varepsilon, \tau\rangle$, where ε is the empty type environment.

It is certainly cumbersome, if not completely impractical, to program with f.o.a.s. trees, and the direct use of deBruijn indices further worsens the situation. To address this issue, we adopt some meta-programming syntax from Scheme and MetaML to facilitate the construction of meta-programs and then provide a translation to eliminate the meta-programming syntax. We also provide interesting examples in support of this design.

[3]For this purpose, we may need to introduce a constructor *HOASvar* of the type $\forall\alpha.string \to (\alpha)HOAS$ for representing free variables.

kinds	κ	::=	$type \mid env$
types	τ	::=	$\alpha \mid \tau_1 \to \tau_2 \mid \langle G, \tau \rangle \mid \forall \overline{\alpha\gamma} : \kappa.\tau$
type env.	G	::=	$\gamma \mid \varepsilon \mid \tau :: G$
constants	c	::=	$cc \mid cf$
const. fun.	cf	::=	run
const. con.	cc	::=	$Lift \mid One \mid Shi \mid App \mid Lam \mid Fix$
expressions	e	::=	$x \mid f \mid c(e_1, \ldots, e_n) \mid$ **lam** $x.e \mid e_1(e_2) \mid$ **fix** $f.e \mid \Lambda^i(v) \mid \Lambda^e(e)$
values	v	::=	$x \mid cc(v_1, \ldots, v_n) \mid$ **lam** $x.e \mid \Lambda^i(v)$
exp. var. ctx.	Γ	::=	$\emptyset \mid \Gamma, x : \tau$
typ. var. ctx.	Δ	::=	$\emptyset \mid \Delta, \alpha : type \mid \Delta, \gamma : env$

Figure 1. The syntax for λ_{code}

The main contribution of the paper lies in recognition and then formalization of a novel approach to typed meta-programming that is practical, general and flexible. This approach makes use of a first-order typeful code representation that not only supports direct open code manipulation but also prevents free variable evaluation. Furthermore, we facilitate meta-programming by providing certain meta-programming syntax as well as a type system to directly support it. The formalization of the type system, which is considerably involved, constitutes the major technical contribution of the paper.

We organize the rest of the paper as follows. In Section 2, we introduce an internal language λ_{code} and use it as the basis for typed meta-programming. We then extend λ_{code} to λ^+_{code} in Section 3, including some syntax to facilitate meta-programming. In Section 4, we briefly mention an external language which is designed for the programmer to construct programs that can eventually be transformed into those in λ^+_{code}. We also present some examples in support of the practicality of meta-programming with λ^+_{code}. In Section 5, we introduce additional code constructors to support more programming features such as references. Lastly, we mention some related work and then conclude.

2 The Language λ_{code}

In this section, we introduce a language λ_{code}, which essentially extends the second-order polymorphic λ-calculus with general recursion (through a fixed point operator **fix**), certain code constructors for constructing typeful code representation and a special function *run* for executing closed code. The syntax of λ_{code} is given in Figure 1. We provide some explanation on the syntax as follows.

- We use the kinds *type* and *env* for types and type environments, respectively. In addition, we use α and γ for the variables ranging over types and type environments, respectively, and $\overline{\alpha\gamma}$ for either an α or a γ.
- We use τ for types and G for type environments. A type environment assigns types to free expression variables in code. For instance, $bool :: int :: \varepsilon$ is a type environment which assigns the types *bool* and *int* to the first and the second expression variables, respectively. We use $\overline{\tau G}$ for either a τ or a G.
- We use $\langle G, \tau \rangle$ as the type for expressions representing code of type τ in which each free variable is assigned a type by the type environment G. For instance, the expression $App(One, Shi(One))$ can be assigned the type $\langle (int \to int) :: int :: \varepsilon, int \rangle$ to indicate that the expression represents some code of type *int* in which there are at most two free variables such that the first and the second free variables have the types *int* and $int \to int$, respectively.
- The (code) constructors $Lift, One, Shi, Lam, App$ and Fix are used for constructing expressions representing typed code in which variables are replaced with deBruijn indices [8], and the function *run* is for executing typed closed code represented by expressions.
- We differentiate **lam**-bound variables x from **fix**-bound variables f; a **lam**-bound variable is a value but a **fix**-bound variable is not. This differentiation mainly prepares for introducing effects into the system.
- We use $\Lambda^i(\cdot)$ and $\Lambda^e(\cdot)$ to indicate type abstraction and application, respectively. For instance, the expression $(\Lambda\alpha.\lambda x : \alpha.x)[int]$ in the Church style is represented as $\Lambda^e(\Lambda^i(\textbf{lam}\ x.x))$. Later, the presence of Λ^i and Λ^e allows us to uniquely determine the rule that is applied last in the typing derivation of a given expression. Preparing for accommodating effects in λ_{code}, we impose the usual value restriction [34] by requiring that Λ^i be only applied to values.

$Lift$	:	$\forall\gamma.\forall\alpha.(\alpha) \Rightarrow \langle \gamma, \alpha \rangle$
Lam	:	$\forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle \alpha_1 :: \gamma, \alpha_2 \rangle) \Rightarrow \langle \gamma, \alpha_1 \to \alpha_2 \rangle$
App	:	$\forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle \gamma, \alpha_1 \to \alpha_2 \rangle, \langle \gamma, \alpha_1 \rangle) \Rightarrow \langle \gamma, \alpha_2 \rangle$
Fix	:	$\forall\gamma.\forall\alpha.(\langle \alpha :: \gamma, \alpha \rangle) \Rightarrow \langle \gamma, \alpha \rangle$
One	:	$\forall\gamma.\forall\alpha.() \Rightarrow \langle \alpha :: \gamma, \alpha \rangle$
Shi	:	$\forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle \gamma, \alpha_1 \rangle) \Rightarrow \langle \alpha_2 :: \gamma, \alpha_1 \rangle$
run	:	$\forall\alpha.(\langle \varepsilon, \alpha \rangle) \Rightarrow \alpha$

Figure 2. The types of some constructors in λ_{code}

- We use $\Lambda^i(\cdot)$ and $\Lambda^e(\cdot)$ to indicate type abstraction and application, respectively. For instance, the expression $(\Lambda\alpha.\lambda x : \alpha.x)[int]$ in the Church style is represented as $\Lambda^e(\Lambda^i(\textbf{lam}\ x.x))$. Later, the presence of Λ^i and Λ^e allows us to uniquely determine the rule that is applied last in the typing derivation of a given expression. Preparing for accommodating effects in λ_{code}, we impose the usual value restriction [34] by requiring that Λ^i be only applied to values.

It is straightforward to extend λ_{code} with some base types (e.g., *bool* and *int* for booleans and integers, respectively) and constants and functions related to these base types. Also, conditional expressions can be readily added into λ_{code}. Later, we may form examples involving these extra features so as to give a more interesting presentation.

We assume a variable can be declared at most once in an expression (type) variable context Γ (Δ). For an expression variable context Γ, we write $\textbf{dom}(\Gamma)$ for the set of variables declared in Γ and $\Gamma(\overline{xf}) = \tau$ if $\overline{xf} : \tau$ is declared in Γ. Note that similar notation also applies to type variable contexts Δ.

We use a signature Σ to assign each constant c a c-type of the following form,

$$\forall \overline{\alpha\gamma}_1 : \kappa_1 \ldots \forall \overline{\alpha\gamma}_m : \kappa_m.(\tau_1, \ldots, \tau_n) \Rightarrow \tau$$

where n indicates the arity of c. We write $c(e_1, \ldots, e_n)$ for applying a constant c of arity n to n arguments $e_1, \ldots, e_n$. For constants c of arity 0, we may write c for $c()$.

For convenience, we may write $\forall\Delta$ for a list of quantifiers $\forall \overline{\alpha\gamma}_1 : \kappa_1 \ldots \forall \overline{\alpha\gamma}_m : \kappa_m$, where

$$\Delta = \emptyset, \overline{\alpha\gamma}_1 : \kappa_1, \ldots, \overline{\alpha\gamma}_m : \kappa_m$$

Also, we may write $\forall\alpha$ and $\forall\gamma$ for $\forall\alpha : type$ and $\forall\gamma : env$, respectively. In Figure 2, we list the c-types assigned to the code constructors and the function *run*. Note that a c-type is *not* regarded as a type.

2.1 Static and Dynamic Semantics

We present the kinding rules for λ_{code} in Figure 3. We use a judgment of the form $\Delta \vdash \tau : type$ ($\Delta \vdash G : env$) to mean that τ (G) is a well-formed type (type environment) under Δ. We use Θ for finite mappings defined below and $\textbf{dom}(\Theta)$ for the domain of Θ.

$$\Theta \quad ::= \quad [] \mid \Theta[\overline{\alpha\gamma} \mapsto \overline{\tau G}]$$

Kinding rules $\Delta \vdash \overline{\tau G} : \kappa$

$$\frac{\Delta(\overline{\alpha\gamma}) = \kappa}{\Delta \vdash \overline{\alpha\gamma} : \kappa}$$

$$\frac{\Delta \vdash \tau_1 : \mathit{type} \quad \Delta \vdash \tau_2 : \mathit{type}}{\Delta \vdash \tau_1 \to \tau_2 : \mathit{type}}$$

$$\frac{\Delta \vdash G : \mathit{env} \quad \Delta \vdash \tau : \mathit{type}}{\Delta \vdash \langle G, \tau \rangle : \mathit{type}}$$

$$\frac{\Delta, \overline{\alpha\gamma} : \kappa \vdash \overline{\tau G} : \mathit{type}}{\Delta \vdash \forall \overline{\alpha\gamma} : \kappa . \overline{\tau G} : \mathit{type}}$$

$$\frac{}{\Delta \vdash \varepsilon : \mathit{env}}$$

$$\frac{\Delta \vdash \tau : \mathit{type} \quad \Delta \vdash G : \mathit{env}}{\Delta \vdash \tau :: G : \mathit{env}}$$

Figure 3. The kinding rules for λ_{code}

Note that [] stands for the empty mapping and $\Theta[\overline{\alpha\gamma} \mapsto \overline{\tau G}]$ stands for the mapping that extends Θ with a link form $\overline{\alpha\gamma}$ to $\overline{\tau G}$, where we assume $\overline{\alpha\gamma} \notin \mathbf{dom}(\Theta)$. We write $\overline{\tau G}[\Theta]$ for the result of substituting each $\overline{\alpha\gamma} \in \mathbf{dom}(\Theta)$ with $\Theta(\overline{\alpha\gamma})$ in $\overline{\tau G}$. The standard definition of substitution is omitted here. We write $\Delta \vdash \Theta : \Delta_0$ to mean that for each $\overline{\alpha\gamma} \in \mathbf{dom}(\Theta) = \mathbf{dom}(\Delta_0)$, $\Delta \vdash \Theta(\overline{\alpha\gamma}) : \Delta_0(\overline{\alpha\gamma})$.

Given a type variable context Δ and an expression variable context Γ, we write $\Delta \vdash \Gamma$ [ok] to mean that $\Delta \vdash \Gamma(x) : \mathit{type}$ is derivable for every $x \in \mathbf{dom}(\Gamma)$. We use $\Delta; \Gamma \vdash e : \tau$ for a typing judgment meaning that the expression e can be assigned the type τ under $\Delta; \Gamma$, where we require $\Delta \vdash \Gamma$ [ok].

The typing rules for λ_{code} are listed in Figure 4. In the rule **(ty-iLam)**, which introduces Λ^i, the premise $\Delta \vdash \Gamma$ [ok] ensures that there are no free occurrences of $\overline{\alpha\gamma}$ in Γ.

PROPOSITION 2.1. *(Canonical Forms) Assume that $\emptyset; \emptyset \vdash v : \tau$ is derivable. Then we have the following.*

- *If $\tau = \tau_1 \to \tau_2$, then v is of the form* **lam** *$x.e$.*
- *If $\tau = \langle G, \tau_1 \rangle$, then v is of one of the following forms: $\mathit{Lift}(v_1)$, One, $\mathit{Shi}(v_1)$, $\mathit{Lam}(v_1)$, $\mathit{App}(v_1, v_2)$ and $\mathit{Fix}(v_1)$.*
- *If $\tau = \forall \overline{\alpha\gamma}.\kappa$, then v is of the form $\Lambda^i(v_1)$.*

PROOF. The proposition follows from a straightforward inspection of the typing rules in Figure 4. □

We use θ for finite mappings defined below:

$$\theta \quad ::= \quad [] \mid \theta[\overline{xf} \mapsto e]$$

and write $e[\theta]$ for the result of substituting each $\overline{xf} \in \mathbf{dom}(\theta)$ for $\theta(\overline{xf})$ in e. We write

$$\Delta; \Gamma \vdash (\Theta; \theta) : (\Delta_0; \Gamma_0)$$

to mean $\Delta \vdash \Theta : \Delta_0$ and for each $\overline{xf} \in \mathbf{dom}(\theta) = \mathbf{dom}(\Gamma_0)$, $\Delta; \Gamma \vdash \theta(\overline{xf}) : \Gamma_0(\overline{xf})[\Theta]$.

We assign dynamic semantics to λ_{code} through the use of evaluation

Typing rules $\Delta; \Gamma \vdash e : \tau$

$$\frac{\Delta \vdash \Gamma \text{ [ok]} \quad \Gamma(\overline{xf}) = \tau}{\Delta; \Gamma \vdash \overline{xf} : \tau} \textbf{ (ty-var)}$$

$$\frac{\Sigma(c) = \forall \Delta_0.(\tau_1, \ldots, \tau_n) \Rightarrow \tau \quad \Delta \vdash \Gamma \text{ [ok]} \quad \Delta \vdash \Theta : \Delta_0 \quad \Delta; \Gamma \vdash e_i : \tau_i[\Theta] \text{ for } i = 1, \ldots, n}{\Delta; \Gamma \vdash c(e_1, \ldots, e_n) : \tau[\Theta]} \textbf{ (ty-cst)}$$

$$\frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \textbf{lam}\ x.e : \tau_1 \to \tau_2} \textbf{ (ty-lam)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1(e_2) : \tau_2} \textbf{ (ty-app)}$$

$$\frac{\Delta; \Gamma, f : \tau \vdash e : \tau}{\Delta; \Gamma \vdash \textbf{fix}\ f.e : \tau} \textbf{ (ty-fix)}$$

$$\frac{\Delta, \overline{\alpha\gamma} : \kappa; \Gamma \vdash v : \tau \quad \Delta \vdash \Gamma \text{ [ok]}}{\Delta; \Gamma \vdash \Lambda^i(v) : \forall \overline{\alpha\gamma} : \kappa . \tau} \textbf{ (ty-iLam)}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \overline{\alpha\gamma} : \kappa . \tau \quad \Delta \vdash \overline{\tau G} : \kappa}{\Delta; \Gamma \vdash \Lambda^e(e) : \tau[\overline{\alpha\gamma} \mapsto \overline{\tau G}]} \textbf{ (ty-eLam)}$$

Figure 4. The typing rules for λ_{code}

contexts, which are defined as follows.

$$\text{eval. ctx.} \quad E \quad ::= \quad [] \mid c(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n) \mid E(e) \mid v(E) \mid \Lambda^e(E)$$

Given an evaluation context E and an expression e, we use $E[e]$ for the expression obtained from replacing the hole [] in E with e.

We define a function *comp* as follows, where we use $\overline{xfs}$ for a sequence of distinct expression variables $\overline{xf}$. Note that *comp* is a function at meta-level.

$$\begin{array}{rcl}
\mathit{comp}(\overline{xfs}; \mathit{Lift}(v)) & = & v \\
\mathit{comp}(\overline{xfs}, \overline{xf}; \mathit{One}) & = & \overline{xf} \\
\mathit{comp}(\overline{xfs}, \overline{xf}; \mathit{Shi}(v)) & = & \mathit{comp}(\overline{xfs}; v) \\
\mathit{comp}(\overline{xfs}; \mathit{Lam}(v)) & = & \textbf{lam}\ x.\mathit{comp}(\overline{xfs}, x; v) \\
\mathit{comp}(\overline{xfs}; \mathit{App}(v_1, v_2)) & = & (\mathit{comp}(\overline{xfs}; v_1))(\mathit{comp}(\overline{xfs}; v_2)) \\
\mathit{comp}(\overline{xfs}; \mathit{Fix}(v)) & = & \textbf{fix}\ f.\mathit{comp}(\overline{xfs}, f; v)
\end{array}$$

Intuitively, when applied to a sequence of distinct expression variables $\overline{xfs}$ and a value v representing some code, *comp* returns the code. For instance, we have

$$\mathit{comp}(\cdot, x, f; \mathit{App}(\mathit{One}, \mathit{Shi}(\mathit{One}))) = f(x).$$

DEFINITION 2.2. *We define redexes and their reductions as follows.*

- *$(\textbf{lam}\ x.e)(v)$ is a redex, and its reduction is $e[x \mapsto v]$.*
- *$\textbf{fix}\ f.e$ is a redex, and its reduction is $e[f \mapsto \textbf{fix}\ f.e]$.*
- *$\Lambda^e(\Lambda^i(v))$ is a redex, and its reduction is v.*
- *$\mathit{run}(v)$ is a redex if $\mathit{comp}(\cdot; v)$ is defined, and its reduction is $\mathit{comp}(\cdot; v)$.*

Given expressions $e = E[e_1]$ and $e' = E[e_1']$, we write $e \to e'$ and say e reduces to e' in one step if e_1 is a redex and e_1' is its reduction.

2.2 Key Properties

We first establish the following substitution lemma.

LEMMA 2.3. *(Substitution) Assume that* $\Delta,\Delta_0;\Gamma,\Gamma_0 \vdash e : \tau$ *is derivable and* $\Delta;\Gamma \vdash (\Theta;\theta) : (\Delta_0;\Gamma_0)$ *holds. Then* $\Delta;\Gamma \vdash e[\theta] : \tau[\Theta]$ *is derivable.*

PROOF. The proof follows from structural induction on the typing derivation of $\Delta,\Delta_0;\Gamma,\Gamma_0 \vdash e : \tau$. □

We now define a function $G(\cdot)$ as follows that maps a given expression variable context to a type environment.

$$G(\emptyset) = \varepsilon \qquad G(\Gamma, x:\tau) = \tau :: G(\Gamma)$$

LEMMA 2.4. *Let* Γ *be* $\overline{xf_1} : \tau_1, \ldots, \overline{xf_n} : \tau_n$. *If* $\emptyset \vdash \Gamma\ [ok]$ *holds and* $\emptyset;\emptyset \vdash v : \langle G(\Gamma), \tau\rangle$ *is derivable, then* $\emptyset;\Gamma \vdash comp(\overline{xf_1}, \ldots, \overline{xf_n}; v) : \tau$ *is derivable.*

PROOF. This follows from structural induction on v. □

THEOREM 2.5. *(Subject Reduction) Assume* $\emptyset;\emptyset \vdash e : \tau$ *is derivable. If* $e \rightarrow e'$ *holds, then* $\emptyset;\emptyset \vdash e' : \tau$ *is derivable.*

PROOF. With Lemma 2.3 and Lemma 2.4, the proof follows from structural induction on the typing derivation of $\emptyset;\emptyset \vdash e : \tau$. □

THEOREM 2.6. *(Progress) Assume* $\emptyset;\emptyset \vdash e : \tau$ *is derivable. Then* e *is either a value or* $e \rightarrow e'$ *holds for some expression* e'.

PROOF. With Proposition 2.1, the theorem follows from structural induction on the typing derivation of $\emptyset;\emptyset \vdash e : \tau$. □

Combining Theorem 2.5 and Theorem 2.6, we clearly have that the evaluation of a well-typed closed expression e in λ_{code} either reaches a value or continues forever. In particular, this indicates that the problem of free variable evaluation can never occur in λ_{code}.

2.3 Meta-Programming with λ_{code}

It is already possible to do meta-programming with λ_{code}. For instance, we can first form an external language ML_{code} by extending ML with code constructors (*Lift*, *One*, *Shi*, *App*, *Lam*, *Fix*) and the special function *run*, and then employ a type inference algorithm (e.g., one based on the one described in [4]) to elaborate programs in ML_{code} into programs, or more precisely typing derivations of programs, in λ_{code}.

As an example, we show that the function *genEvalPoly* in Section 1 can be implemented as follows, where we use `[]` for empty type environment ε and `<;>` for the type constructor $\langle\cdot,\cdot\rangle$.

```
val plus = fn x: int => fn y: int => x + y
val mult = fn x: int => fn y: int => x * y
fun genEvalPoly (p) =
  let
      fun aux (p) =
        if null (p) then Lift (0)
        else App (App (Lift plus, Lift (hd p)),
                  App (App (Lift mult, One),
                       aux (tl p)))
      withtype int list -> <int :: []; int>
  in
      Lam (aux p)
  end
withtype int list -> <[]; int -> int>
```

The `withtype` clause following the definition of the function *aux* is a type annotation indicating that *aux* expects to be assigned the type

```
typecon (type, type) FOAS =
  {'g,'a}. ('g,'a) FOASlift of 'a
| {'g,'a}. ('a * 'g, 'a) FOASone
| {'g,'a1,'a2}.
    ('a1 * 'g, 'a2) FOASshi of ('g, 'a2) FOAS
| {'g,'a1,'a2}.
    ('g, 'a1 -> 'a2) FOASlam of ('a1 * 'g, 'a2) FOAS
| {'g,'a1,'a2}.
    ('g, 'a2) FOASapp of
      ('g, 'a1 -> 'a2) FOAS * ('g, 'a1) FOAS
| {'g, 'a}. ('g, 'a) FOASfix of ('a * 'g, 'a) FOAS

typecon (type) ENV =
  (unit) ENVnil
| {'g,'a}. ('a * 'g) ENVcons of 'a * ('g) ENV

(* 'fix x => e' is the fixed point of 'fn x => e' *)
fun comp (FOASlift v) = (fn env => v)
  | comp (FOASone) = (fn (ENVcons (v, _)) => v)
  | comp (FOASshi e) = let
      val c = comp e
    in fn (ENVcons (_, env)) => c env end
  | comp (FOASlam e) = let
      val c = comp e
    in fn env => fn v => c (ENVcons (v, env)) end
  | comp (FOASapp (e1, e2)) = let
      val c1 = comp e1
      val c2 = comp e2
    in fn env => (c1 env) (c2 env) end
  | comp (FOASfix e) = let
      val c = comp e
    in fn env => fix v => c (ENVcons (v, env) end
withtype {'g,'a}. ('g,'a) FOAS -> ('g ENV -> 'a)

fun run e = (comp e) (ENVnil)
withtype {'a}. (unit, 'a) FOAS -> 'a
```

Figure 5. Implementing code constructors and *run*

$(int)list \rightarrow \langle int :: \varepsilon, int\rangle$, that is, *aux* takes an integer list and returns some code of type *int* in which the first and only free variable has type *int*. Similarly, the `withtype` clause for *genEvalPoly* means that *genEvalPoly* takes an integer list and returns some closed code of type $int \rightarrow int$.

Given the obvious meaning of *null*, *hd* and *tl*, it should be straightforward to relate the ML-like concrete syntax used in the above program to the syntax of (properly extended) λ_{code}. Evidently, this kind of programming style is at least unwieldy if not impractical. To some extent, this is just like writing meta-programs in Scheme without using the backquote/comma notation. Therefore, we are naturally led to provide some syntactic support to facilitate meta-programming.

2.4 Embedding λ_{code} into $\lambda_{2,G\mu}$

Before presenting syntactic support for meta-programming, we show a direct embedding of λ_{code} in $\lambda_{2,G\mu}$, where $\lambda_{2,G\mu}$ is an internal language that essentially extends the second order polymorphic λ-calculus with guarded recursive (g.r.) datatype constructors [35]. This simple and interesting embedding, which the reader can skip without affecting the understanding of the rest of the paper, indicates that the code constructors in λ_{code} can be readily interpreted through g.r. datatypes.

In Figure 5, we use some concrete syntax of $\mathrm{ML}_{2,G\mu}$ to declare a binary g.r. datatype constructor $(\cdot,\cdot)$*FOAS*, where $\mathrm{ML}_{2,G\mu}$ [35] is an external language of $\lambda_{2,G\mu}$. The code constructors *Lift*, *One*, *Shi*, *App*, *Lam*, *Fix* have their counterparts *FOASlift*, *FOAone*, *FOAshi*, *FOAapp*, *FOASlam*, *FOASfix* in $\lambda_{2,G\mu}$.

We use a type in $\lambda_{2,G\mu}$ for representing a type environment in λ_{code}; the unit type $\mathbf{1}$ represents the empty type environment ε, and the type constructor $*$, which is for constructing product types, represents the type environment constructor ::; the type constructor $(\cdot,\cdot)$*FOAS* represents $\langle\cdot,\cdot\rangle$. Formally, we define a translation $|\cdot|$ as follows, which translates type environments and types in λ_{code} into types in $\lambda_{2,G\mu}$.

$$\begin{array}{rcl} |\varepsilon| & = & \mathbf{1} \\ |\tau :: G| & = & |\tau| * |G| \\ |\overline{\alpha\gamma}| & = & \overline{\alpha\gamma} \\ |\tau_1 \to \tau_2| & = & |\tau_1| \to |\tau_2| \\ |\langle G, \tau\rangle| & = & (|G|, |\tau|)\mathit{FOAS} \\ |\forall\overline{\alpha\gamma} : \kappa.\tau| & = & \forall\overline{\alpha\gamma}.|\tau| \end{array}$$

The function *run* is implemented in Figure 5. We use a `withtype` clause for introducing a type annotation. The type annotation for *run* indicates that *run* is expected to be assigned the type $\forall\alpha.(\mathbf{1},\alpha)\mathit{FOAS} \to \alpha$, which corresponds to the type $\forall\alpha.\langle\varepsilon,\alpha\rangle \to \alpha$ in λ_{code}. However, it needs to be pointed out that this implementation of *run* cannot support run-time code generation, for which we need a (primitive) function that can perform compilation at run-time and then upload the code generated from the compilation.

With this embedding of λ_{code} in $\lambda_{2,G\mu}$, we are able to construct programs for performing analysis on typeful code representation. For instance, the function *comp* defined in Figure 5 is such an example.

3 The Language λ^{+}_{code}

We extend λ_{code} to λ^{+}_{code} with some meta-programming syntax adopted from Scheme and MetaML.

$$\text{expressions} \quad e \quad ::= \quad \ldots \mid \text{`}(e) \mid \hat{}(e)$$

Loosely speaking, the notation $\text{`}(\cdot)$ corresponds to the backquote notation in Scheme (or the notation $\langle\cdot\rangle$ in MetaML), and we use $\text{`}(e)$ as the code representation for e. On the other hand, $\hat{}(\cdot)$ corresponds to the comma notation in Scheme (or the notation $\tilde{}(\cdot)$ in MetaML), and we use $\hat{}(e)$ for splicing the code e into some context. We refer $\text{`}(\cdot)$ and $\hat{}(\cdot)$ as meta-programming syntax.

The expression variable context Γ is now defined as follows,

$$\text{exp. var. ctx.} \quad \Gamma \quad ::= \quad \emptyset \mid \Gamma, \overline{xf}@k : \tau$$

where $\overline{xf}@k$ stands for variables at level $k \geq 0$ and we use the name *staged variable* for $\overline{xf}@k$. Intuitively, an expression e in the empty evaluation context is said to be at level 0; if an occurrence of e in e_0 is at level k, then the occurrence of e in $\text{`}(e_0)$ is at level $k+1$; if an occurrence of e in e_0 is at level $k+1$, then the occurrence of e in $\hat{}(e_0)$ is at level k; if an occurrence of $\mathbf{lam}\ x.e_1$ or $\mathbf{fix}\ f.e_1$ is at level k, then x or f is bound at level k. A declared staged variable $\overline{xf}@k$ in Γ simply indicates that $\overline{xf}$ is to be bound at level k.

3.1 Static Semantics

For each natural number k, let $\mathbf{pos}_k$ be the set $\{1,\ldots,k\}$, or formally $\{n \mid 0 < n \leq k\}$. We use $\mathcal{G}$ for finite mappings from positive integers to type environments such that the domains of $\mathcal{G}$ are always equal to $\mathbf{pos}_k$ for some k. In particular, we use $\emptyset$ for the mapping $\mathcal{G}$ such that $\mathbf{dom}(\mathcal{G}) = \mathbf{pos}_0$. We write $\Delta \vdash^{\mathcal{G}}_k \Gamma\ [\mathrm{ok}]$ to mean that

Typing rules $\quad \Delta;\Gamma \vdash^{\mathcal{G}}_k e : \tau$

$$\frac{\Delta \vdash^{\mathcal{G}}_k \Gamma\ [\mathrm{ok}] \quad \Gamma(\overline{xf}@0) = \tau}{\Delta;\Gamma \vdash^{\mathcal{G}}_k \overline{xf} : \tau}\ \textbf{(ty-var-0)}$$

$$\frac{\Delta \vdash^{\mathcal{G}}_{k+1} \Gamma\ [\mathrm{ok}] \quad \Gamma(\overline{xf}@(k+1)) = \tau}{\Delta;\Gamma \vdash^{\mathcal{G}}_{k+1} \overline{xf} : \tau}\ \textbf{(ty-var-1)}$$

$$\frac{\Sigma(c) = \forall\Delta_0.(\tau_1,\ldots,\tau_n) \Rightarrow \tau \quad \Delta \vdash^{\mathcal{G}}_k \Gamma\ [\mathrm{ok}] \quad \Delta \vdash \Theta : \Delta_0 \quad \Delta;\Gamma \vdash^{\mathcal{G}}_k e_i : \tau_i[\Theta]\ \text{ for } i = 1,\ldots,n}{\Delta;\Gamma \vdash^{\mathcal{G}}_k c(e_1,\ldots,e_n) : \tau[\Theta]}\ \textbf{(ty-cst)}$$

$$\frac{\Delta;\Gamma, x@k : \tau_1 \vdash^{\mathcal{G}}_k e : \tau_2}{\Delta;\Gamma \vdash^{\mathcal{G}}_k \mathbf{lam}\ x.e : \tau_1 \to \tau_2}\ \textbf{(ty-lam)}$$

$$\frac{\Delta;\Gamma \vdash^{\mathcal{G}}_k e_1 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash^{\mathcal{G}}_k e_2 : \tau_1}{\Delta;\Gamma \vdash^{\mathcal{G}}_k e_1(e_2) : \tau_2}\ \textbf{(ty-app)}$$

$$\frac{\Delta;\Gamma, f@k : \tau \vdash^{\mathcal{G}}_k e : \tau}{\Delta;\Gamma \vdash^{\mathcal{G}}_k \mathbf{fix}\ f.e : \tau}\ \textbf{(ty-fix)}$$

$$\frac{\Delta;\Gamma \vdash^{\mathcal{G}+G}_{k+1} e : \tau}{\Delta;\Gamma \vdash^{\mathcal{G}}_k \text{`}(e) : \langle G(k+1;\Gamma), \tau\rangle}\ \textbf{(ty-encode)}$$

$$\frac{\Delta;\Gamma \vdash^{\mathcal{G}}_k e : \langle G(k+1;\Gamma), \tau\rangle}{\Delta;\Gamma \vdash^{\mathcal{G}+G}_{k+1} \hat{}(e) : \tau}\ \textbf{(ty-decode)}$$

$$\frac{\Delta, \overline{\alpha\gamma} : \kappa;\Gamma \vdash^{\emptyset}_0 v : \tau \quad \Delta \vdash^{\emptyset}_0 \Gamma\ [\mathrm{ok}]}{\Delta;\Gamma \vdash^{\emptyset}_0 \Lambda^{i}(v) : \forall\overline{\alpha\gamma} : \kappa.\tau}\ \textbf{(ty-iLam)}$$

$$\frac{\Delta;\Gamma \vdash^{\emptyset}_0 e : \forall\overline{\alpha\gamma} : \kappa.\tau \quad \Delta \vdash \overline{\tau G} : \kappa}{\Delta;\Gamma \vdash^{\emptyset}_0 \Lambda^{e}(e) : \tau[\overline{\alpha\gamma} \mapsto \overline{\tau G}]}\ \textbf{(ty-eLam)}$$

Figure 6. The typing rules for λ^{+}_{code}

1. $\Delta \vdash \Gamma(\overline{xf}) : \mathit{type}$ for each $\overline{xf} \in \mathbf{dom}(\Gamma)$, and
2. $\mathbf{dom}(\mathcal{G}) = \mathbf{pos}_k$, and
3. $\Delta \vdash \mathcal{G}(n) : \mathit{env}$ for each $n \in \mathbf{dom}(\mathcal{G})$.

In addition, we introduce the following definitions.

- Given G, $k > 0$ and Γ, we define $G(k;\Gamma)$ as follows.

$$\begin{array}{rcll} G(k;\emptyset) & = & G & ; \\ G(k;\Gamma, \overline{xf}@n : \tau) & = & \tau :: G(k;\Gamma) & \text{if } n = k; \\ G(k;\Gamma, \overline{xf}@n : \tau) & = & G(k;\Gamma) & \text{if } n \neq k. \end{array}$$

- Given $\mathcal{G}$, Γ and τ, we define $\mathcal{G}(0;\Gamma;\tau) = \tau$ and

$$\mathcal{G}(k;\Gamma;\tau) = \mathcal{G}(k-1;\Gamma;\langle G(k;\Gamma), \tau\rangle)$$

  for $k \in \mathbf{dom}(\mathcal{G})$, where $G = \mathcal{G}(k)$. We write $\mathcal{G}(\Gamma;\tau)$ for $\mathcal{G}(k;\Gamma;\tau)$ if $\mathbf{dom}(\mathcal{G}) = \mathbf{pos}_k$.

- Given $\mathcal{G}$ and G such that $\mathbf{dom}(\mathcal{G}) = \mathbf{pos}_k$, we use $\mathcal{G} + G$ for the mapping $\mathcal{G}_1$ such that $\mathbf{dom}(\mathcal{G}_1) = \mathbf{pos}_{k+1}$, $\mathcal{G}_1(n) = \mathcal{G}(n)$ for each $n \in \mathbf{pos}_k$ and $\mathcal{G}_1(k+1) = G$.

We use $\Delta;\Gamma \vdash_k^{\mathcal{G}} e : \tau$ for a typing judgment in λ^+_{code}, where we require $\Delta \vdash_k^{\mathcal{G}} \Gamma$ [ok]. Intuitively, $\mathcal{G}(k)$ stands for the initial type environment for code at level k. We present the typing rules for λ^+_{code} in Figure 6. Note that polymorphic code is only allowed to occur at level 0.

3.2 Translation from λ^+_{code} into λ_{code}

We introduce some notations needed in the following presentation. For $n \geq 0$, we use $\Lambda^i_n(e)$ for $\Lambda^i(\ldots(\Lambda^i(e))\ldots)$, where there are n occurrences of Λ^i, and we use $\Lambda^e_n(e)$ similarly. Also, we now use $\overline{xfs}$ for a sequence of staged variables, that is, $\overline{xfs}$ is of the form $\overline{xf}_1@k_1,\ldots,\overline{xf}_n@k_n$. For each $k > 0$, we define $var_k(\overline{xfs};\overline{xf})$ as follows under the assumption that $\overline{xf}@k$ occurs in $\overline{xfs}$: for $\overline{xfs} = (\overline{xfs}_1,\overline{xf}_1@k_1)$, $var_k(\overline{xfs};\overline{xf})$ is

$$\begin{array}{ll} \Lambda^e_{k+1}(One_k) & \text{if } k_1 = k \text{ and } \overline{xf}_1 = \overline{xf}; \\ \Lambda^e_{k+2}(Shi_k)(var_k(\overline{xfs}_1;\overline{xf})) & \text{if } k_1 = k \text{ and } \overline{xf}_1 \neq \overline{xf}; \\ var_k(\overline{xfs}_1;\overline{xf}) & \text{if } k_1 \neq k \end{array}$$

In Figure 7, we define a translation $trans_k(\cdot;\cdot)$ for each $k \geq 0$ that translates expressions in λ^+_{code} into those in λ_{code}. We first define some functions that are needed in the definition of $trans_k(\cdot;\cdot)$. These functions basically generalize the code constructors we have. Given $e, e_1, \ldots, e_n$, we write $Lift^n(e)$ for $Lift(\ldots(Lift(e))\ldots)$, where there are n occurrences of $Lift$; and $App^n(e)(e_1)\ldots(e_n)$ for e if $n = 0$, or for

$$App(App^{n-1}(e)\ldots(e_{n-1}),e_n)$$

if $n > 0$; and $App^n_k(e)(e_1)\ldots(e_n)$ for e if $n = 0$, or for

$$\Lambda^e_{k+2}(App_k)(App^{n-1}_k(e)\ldots(e_{n-1}))(e_n)$$

if $n > 0$. Given type environments $G_1,\ldots,G_n$ and type τ, we write $\langle G_1,\ldots,G_n;\tau\rangle$ for $\langle G_1,\langle\ldots,\langle G_n,\tau\rangle\ldots\rangle\rangle$. With this notation, we have $\mathcal{G}(\Gamma;\tau) = \langle G_1(1;\Gamma),\ldots,G_k(k,\Gamma);\tau\rangle$, where $\mathcal{G} = \emptyset + G_1 + \ldots + G_k$.

A crucial property of $trans_k(\cdot;\cdot)$ is captured by the following lemma, which consists of the main technical contribution of the paper.

LEMMA 3.1. *Assume that $\Delta;\Gamma \vdash_k^{\mathcal{G}} e : \tau$ is derivable in λ^+_{code} and Γ is of the form $\overline{xf}_1@k_1 : \tau_1,\ldots,\overline{xf}_n@k_n : \tau_n$. Then $\Delta;(\Gamma)_0 \vdash trans_k(\overline{xf}_1@k_1,\ldots,\overline{xf}_n@k_n;e) : \mathcal{G}(\Gamma;\tau)$ is derivable in λ_{code}, where $(\Gamma)_0$ is defined as follows:*

$$\begin{array}{rcl} (\emptyset)_0 & = & \emptyset \\ (\Gamma,x@0:\tau) & = & (\Gamma)_0,x:\tau \\ (\Gamma,x@(k+1):\tau)_0 & = & (\Gamma)_0 \end{array}$$

PROOF. The proof follows from structural induction on the typing derivation of $\Delta;\Gamma \vdash_k^{\mathcal{G}} e : \tau$. □

Given an expression e in λ^+_{code}, we write $trans(e)$ for $trans_0(\emptyset;e)$ (if it is well-defined) and call it the translation of e.

THEOREM 3.2. *Assume that $\emptyset;\emptyset \vdash_0^{\emptyset} e : \tau$ is derivable. Then $\emptyset;\emptyset \vdash trans(e) : \tau$ is derivable.*

PROOF. This immediately follows from Lemma 3.1. □

The programmer can now construct a meta-program in λ^+_{code} that may (and probably should) make use of meta-programming syntax and then assign it the dynamic semantics of its translation in λ_{code}. In other words, we may just treat meta-programming syntax as mere syntactic sugar. This is precisely the significance of Theorem 3.2.

We conclude this section with an example to show how the type system of λ^+_{code} can prevent free variable evaluation. Let us recall the example: `<fn x => ~(run <x>)>` in MetaML, whose evaluation leads to free variable evaluation. In λ^+_{code}, the example corresponds to $e = \text{`}(\textbf{lam}\ x.\hat{}(run(\text{`}(x))))$. Clearly, $trans(e) = trans_0(\emptyset;e) = Lam(run(One))$. Note that the type of the expression *One* must equal $\langle\tau :: G,\tau\rangle$ for some G and τ but *run* is only allowed to be applied to an expression whose type is $\langle\varepsilon,\tau\rangle$ for some τ. Therefore, $trans(e)$ is ill-typed. By Theorem 3.2, e is also ill-typed in λ^+_{code} and thus should be rejected.

3.3 Some Remarks

We mention a few subtle issues so as to facilitate the understanding of λ^+_{code}.

Bound Variables at Stage $k > 0$ At level k for some $k > 0$, a bound variable merely represents a deBruijn index and a binding may vanish or occur "unexpectedly". For instance, let e be the expression $\text{`}(\textbf{lam}\ x.\hat{}(f\ \text{`}x))$ and $e' = trans(e) = Lam(f(One))$.

- Let f be the identity function. Then e' evaluates to $Lam(One)$, which represents the code **lam** $x.x$.
- Let f be the shift function **lam** $x.Shi(x)$. Then e' evaluates to $Lam(Shi(One))$, which represents the code **lam** $x.y$ for some free variable y that is distinct from x; there is no binding between *Lam* and *One* in e'.
- Let f be the lift function **lam** $x.Lift(x)$. Then e' evaluates to $Lam(Lift(One))$ and $run(e')$ evaluates to **lam** $x.One$ (not to **lam** $x.Lift(x)$); there is no "expected" binding between *Lam* and *One* in e'. Let e_0 be the expression $run(run(e')(1))$. Then e_0 is rejected as the expression $run(e')(1)$, which evaluates to *One*, cannot be assigned a type of the form $\langle\varepsilon,\tau\rangle$.[4]

Cross-Stage Persistence In meta-programming, a situation often arises where a value defined at an early stage needs to be used at a later stage. For instance, in the expression $\text{`}(\textbf{lam}\ x.x + x)$, the function $+$, which is defined at stage 0, is used at stage 1. This is called cross-stage persistence (CSP) [32]. As is indicated in the typing rules **(ty-var-0)** and **(ty-cst)**, CSP for variables at stage 0 and constants is implicit in λ^+_{code}. However, for variables introduced at stage $k > 0$, CSP needs to be explicit. For instance, $\text{`}(\textbf{lam}\ x.\text{`}(\textbf{lam}\ y.y(x)))$ is ill-typed in λ^+_{code} as the variable x is introduced at stage 1 but used at stage 2. To make it typable, the programmer needs to insert % in front of x: $\text{`}(\textbf{lam}\ x.\text{`}(\textbf{lam}\ y.y(\%x)))$, where % is a shorthand for $\hat{}Lift$, that is, $\%(e)$ represents $\hat{}(Lift(e))$ for any expression e. Note that *Lift* can also be defined as $\text{`}\%$, that is, $Lift(e)$ can be treated as $\text{`}(\%e)$ for any expression e.

[4] In $\nu^{\Box}$ [21], e_0 cannot be typed, either. However, e_0 can be typed in the current implementation of MetaML [29] and MetaOCaml [30]; in the former e_0 evaluates to 1 (which we suspect may be caused by an implementation error) but in the latter the evaluation of e_0 raises a run-time exception caused by free variable evaluation.

$$
\begin{array}{rcl}
\mathit{Lift}_n & : & \forall\gamma_1\ldots\forall\gamma_n.\forall\alpha.\alpha \to \langle\gamma_1,\ldots,\gamma_n;\alpha\rangle \\
\mathit{Lift}_n & = & \Lambda^i_{n+1}(\mathbf{lam}\ x.\mathit{Lift}^n(x)) \\
\\
\mathit{Lam}_n & : & \forall\gamma_1\ldots\forall\gamma_n.\forall\alpha_1.\forall\alpha_2.\langle\gamma_1,\ldots,\gamma_{n-1},\alpha_1 :: \gamma_n;\alpha_2\rangle \to \langle\gamma_1,\ldots,\gamma_n;\alpha_1\to\alpha_2\rangle \\
\mathit{Lam}_1 & = & \Lambda^i_3(\mathbf{lam}\ x.\mathit{Lam}(x)) \\
\mathit{Lam}_{n+1} & = & \Lambda^i_{n+3}(\mathbf{lam}\ x.\mathit{App}(\mathit{Lift}(\Lambda^e_{n+2}(\mathit{Lam}_n)),x)) \\
\\
\mathit{App}_n & : & \forall\gamma_1\ldots\forall\gamma_n.\forall\alpha_1.\forall\alpha_2.\langle\gamma_1,\ldots,\gamma_n;\alpha_1\to\alpha_2\rangle \to \langle\gamma_1,\ldots,\gamma_n;\alpha_1\rangle \to \langle\gamma_1,\ldots,\gamma_n;\alpha_2\rangle \\
\mathit{App}_1 & = & \Lambda^i_3(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.\mathit{App}(x_1,x_2)) \\
\mathit{App}_{n+1} & = & \Lambda^i_{n+3}(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.\mathit{App}(\mathit{App}(\mathit{Lift}(\Lambda^e_{n+2}(\mathit{App}_n)),x_1),x_2)) \\
\\
\mathit{Fix}_n & : & \forall\gamma_1\ldots\forall\gamma_n.\forall\alpha.\langle\gamma_1,\ldots,\gamma_{n-1},\alpha :: \gamma_n;\alpha\rangle \to \langle\gamma_1,\ldots,\gamma_n;\alpha\to\alpha\rangle \\
\mathit{Fix}_1 & = & \Lambda^i_2(\mathbf{lam}\ x.\mathit{Fix}(x)) \\
\mathit{Fix}_{n+1} & = & \Lambda^i_{n+2}(\mathbf{lam}\ x.\mathit{App}(\mathit{Lift}(\Lambda^e_{n+1}(\mathit{Fix}_n)),x)) \\
\\
\mathit{One}_n & : & \forall\gamma_1\ldots\forall\gamma_n.\forall\alpha.\langle\gamma_1,\ldots,\gamma_{n-1},\alpha :: \gamma_n;\alpha\rangle \\
\mathit{One}_1 & = & \Lambda^i_2(\mathit{One}) \\
\mathit{One}_{n+1} & = & \Lambda^i_{n+2}(\mathit{Lift}(\Lambda^e_{n+1}(\mathit{One}_n))) \\
\\
\mathit{Shi}_n & : & \forall\gamma_1\ldots\forall\gamma_n.\forall\alpha_1.\forall\alpha_2.\langle\gamma_1,\ldots,\gamma_n;\alpha_2\rangle \to \langle\gamma_1,\ldots,\gamma_{n-1},\alpha_1 :: \gamma_n;\alpha_2\rangle \\
\mathit{Shi}_1 & = & \Lambda^i_3(\mathbf{lam}\ x.\mathit{Shi}(x)) \\
\mathit{Shi}_{n+1} & = & \Lambda^i_{n+3}(\mathit{App}(\mathit{Lift}(\Lambda^e_{n+2}(\mathit{Shi}_n)),x))
\end{array}
$$

$\mathit{trans}_0(\cdot;\cdot)$

$$
\begin{array}{rcl}
\mathit{trans}_0(\overline{xfs};\overline{xf}) & = & \overline{xf} \quad \text{if } \overline{xf}@0 \text{ occurs in } \overline{xfs} \\
\mathit{trans}_0(\overline{xfs};c(e_1,\ldots,e_n)) & = & c(\mathit{trans}_0(\overline{xfs};e_1),\ldots,\mathit{trans}_0(\overline{xfs};e_n)) \\
\mathit{trans}_0(\overline{xfs};\mathbf{lam}\ x.e) & = & \mathbf{lam}\ x.\mathit{trans}_0(\overline{xfs},x@0;e) \\
\mathit{trans}_0(\overline{xfs};e_1(e_2)) & = & \mathit{trans}_0(\overline{xfs};e_1)(\mathit{trans}_0(\overline{xfs};e_1)) \\
\mathit{trans}_0(\overline{xfs};\mathbf{fix}\ f.e) & = & \mathbf{fix}\ f.\mathit{trans}_0(\overline{xfs},f@0;e) \\
\mathit{trans}_0(\overline{xfs};\Lambda^i(e)) & = & \Lambda^i(\mathit{trans}_0(\overline{xfs};e)) \\
\mathit{trans}_0(\overline{xfs};\Lambda^e(e)) & = & \Lambda^e(\mathit{trans}_0(\overline{xfs};e)) \\
\mathit{trans}_0(\overline{xfs};\text{`}(e)) & = & \mathit{trans}_1(\overline{xfs};e)
\end{array}
$$

$\mathit{trans}_1(\cdot;\cdot)$

$$
\begin{array}{rcl}
\mathit{trans}_1(\overline{xfs};\overline{xf}) & = & \mathit{Lift}(\overline{xf}) \quad \text{if } \overline{xf}@0 \text{ occurs in } \overline{xfs} \\
\mathit{trans}_1(\overline{xfs};\overline{xf}) & = & \mathit{var}_1(\overline{xfs};\overline{xf}) \quad \text{if } \overline{xf}@1 \text{ occurs in } \overline{xfs} \\
\mathit{trans}_1(\overline{xfs};c(e_1,\ldots,e_n)) & = & \mathit{App}^n(\mathit{Lift}(\mathbf{lam}\,x_1\ldots\mathbf{lam}\,x_n.c(x_1,\ldots,x_n)))(\mathit{trans}_1(\overline{xfs};e_1))\ldots(\mathit{trans}_1(\overline{xfs};e_n)) \\
\mathit{trans}_1(\overline{xfs};\mathbf{lam}\ x.e) & = & \mathit{Lam}(\mathit{trans}_1(\overline{xfs},x@1;e) \\
\mathit{trans}_1(\overline{xfs};e_1(e_2)) & = & \mathit{App}(\mathit{trans}_1(\overline{xfs};e_1),\mathit{trans}_1(\overline{xfs};e_2)) \\
\mathit{trans}_1(\overline{xfs};\mathbf{fix}\ f.e) & = & \mathit{Fix}(\mathit{trans}_1(\overline{xfs},f@1;e)) \\
\mathit{trans}_1(\overline{xfs};\text{`}(e)) & = & \mathit{trans}_2(\overline{xfs};e) \\
\mathit{trans}_1(\overline{xfs};\hat{}(e)) & = & \mathit{trans}_0(\overline{xfs};e)
\end{array}
$$

$\mathit{trans}_k(\cdot;\cdot)$ for $k > 1$

$$
\begin{array}{rcl}
\mathit{trans}_k(\overline{xfs};\overline{xf}) & = & \mathit{Lift}^k(\overline{xf}) \quad \text{if } \overline{xf}@0 \text{ occurs in } \overline{xfs} \\
\mathit{trans}_k(\overline{xfs};\overline{xf}) & = & \mathit{var}_k(\overline{xfs};\overline{xf}) \quad \text{if } \overline{xf}@k \text{ occurs in } \overline{xfs} \\
\mathit{trans}_k(\overline{xfs};c(e_1,\ldots,e_n)) & = & \mathit{App}^n_k(\mathit{Lift}^k(\mathbf{lam}\,x_1\ldots\mathbf{lam}\,x_n.c(x_1,\ldots,x_n)))(\mathit{trans}_k(\overline{xfs};e_1))\ldots(\mathit{trans}_k(\overline{xfs};e_n)) \\
\mathit{trans}_k(\overline{xfs};\mathbf{lam}\ x.e) & = & \Lambda^e_{k+2}(\mathit{Lam}_k)(\mathit{trans}_k(\overline{xfs},x@k;e) \\
\mathit{trans}_k(\overline{xfs};e_1(e_2)) & = & \Lambda^e_{k+2}(\mathit{App}_k)(\mathit{trans}_k(\overline{xfs};e_1))(\mathit{trans}_k(\overline{xfs};e_2)) \\
\mathit{trans}_k(\overline{xfs};\mathbf{fix}\ f.e) & = & \Lambda^e_{k+1}(\mathit{Fix}_k)(\mathit{trans}_k(\overline{xfs},f@k;e) \\
\mathit{trans}_k(\overline{xfs};\text{`}(e)) & = & \mathit{trans}_{k+1}(\overline{xfs};e) \\
\mathit{trans}_k(\overline{xfs};\hat{}(e)) & = & \mathit{trans}_{k-1}(\overline{xfs};e)
\end{array}
$$

Figure 7. The definition of $\mathit{trans}_k(\cdot;\cdot)$ for $k \geq 0$

4 Meta-Programming with λ^{+}_{code}

We now need an external language $\mathrm{ML}^{+}_{\mathrm{code}}$ for the programmer to construct meta-programs and then a process to translate such programs into typing derivations in (properly extended) λ^{+}_{code}. We present one possible design of $\mathrm{ML}^{+}_{\mathrm{code}}$ as follows, where b is for base types such as *bool*, *int*, etc.

types	τ	::=	$b \mid \alpha \mid \tau \to \tau \mid \langle G, \tau \rangle$
type env.	G	::=	$\gamma \mid \varepsilon \mid \tau :: G$
type schemes	σ	::=	$\tau \mid \forall\alpha.\sigma$
expressions	e	::=	$x \mid f \mid c(e_1,\ldots,e_n) \mid \mathbf{if}(e_1,e_2,e_3) \mid$
			$\mathbf{lam}\, x.e \mid \mathbf{lam}\, x:\tau.e \mid e_1(e_2) \mid$
			$\mathbf{fix}\, f.e \mid \mathbf{fix}\, f[\alpha_1,\ldots,\alpha_n]:\tau.e \mid$
			$\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2\, \mathbf{end} \mid (e:\tau)$
			$`(e) \mid \hat{}(e)$

The only unfamiliar syntax is $\mathbf{fix}\, f[\alpha_1,\ldots,\alpha_n]:\tau.e$, which we use to support polymorphic recursion; this expression is expected to be assigned the type scheme $\sigma = \forall\alpha_1\ldots\forall\alpha_n.\tau$.

We have implemented a type inference algorithm based on the one in [4] that supports the usual let-polymorphism. Like in Haskell [22], if the type of a recursive function is given, then polymorphic recursion is allowed in the definition of the function.

We are now ready to present some examples of meta-programs in $\mathrm{ML}^{+}_{\mathrm{code}}$.

Example 1 The previously defined function *genEvalPoly* can now be implemented as follows, which makes no explicit use of code constructors.

```
fun genEvalPoly (p) =
  let
      fun aux p x =
        if null (p) then Lift (0)
        else `(hd p + ^x * ^(aux (tl p) x))
      withtype
        {'g}. int list -> <'g; int> -> <'g; int>
  in
      `(fn x => ^(aux p `x))
  end
withtype {'g}. int list -> <'g; int -> int>
```

Note that the type annotations, which can be automatically inferred, are presented solely for making the program easier to understand.

Example 2 The following program implements the Ackermann function.

```
fun ack m n =
  if m = 0 then n + 1
  else if n = 0 then ack (m-1) 1
       else ack (m-1) (ack m (n-1))
withtype int -> int -> int
```

We can now define a function *genAck* as follows such that the function returns the code for computing $ack(m)$ when applied to a given integer m.

```
fun genAck m =
  if m = 0 then `(fn n => n+1)
  else
    `(fun f (n) =>
      let
          val f' = ^(genAck (m-1))
      in
          if n = 0 then f' 1 else f' (f (n-1))
      end)
withtype {'g}. int -> <'g; int -> int>
```

We use the syntax `` `(fun f (n) => ...) `` for

```
`(fix f => (fn n => ...))',
```

which translates into something of the form $Fix(Lam(...))$. This shows an interesting use of recursion at level 1. Also, we point out that polymorphic recursion is required in this example.[5]

Example 3 We contrast an unstaged implementation of inner product (*innerProd*) with a staged implementation of inner product (*genInnerProd*) in Figure 8. Given a natural number n, $genInnerProd(n)$ returns the code for some function f_1; given an integer vector v_1 of length n, f_1 returns the code for some function f_2; given an integer vector v_2 of length n, f_2 returns the inner product of v_1 and v_2. For instance, if $n = 2$, then f_1 is basically equivalent to the function defined below;

```
fn v1 => `(fn v2 =>
  0 + ^(Lift (sub (v1, 0))) * sub (v2, 0)
    + ^(Lift (sub (v1, 1))) * sub (v2, 1))
```

if $v_1[0] = 6$ and $v_1[1] = 23$, then f_2 is basically equivalent to the function defined below.

```
fn v2 => 0 + 6 * sub (v2, 0) + 23 * sub (v2, 1)
```

Notice that this example involves expressions at level 2 and the use of a CSP operator %.

5 Extensions

It is straightforward to extend λ_{code} (and subsequently λ^{+}_{code}) to support additional language features such as conditionals, pairs, references, etc. For instance, in order to support conditional expressions of the form $\mathbf{if}(e_1,e_2,e_3)$, we introduce a code constructor *If* and assign it the following type;

$$\forall\gamma\forall\alpha.\langle\gamma,bool\rangle \to \langle\gamma,\alpha\rangle \to \langle\gamma,\alpha\rangle \to \langle\gamma,\alpha\rangle$$

we then define If_n as we have done for the previous code constructors in Figure 7, and extend the definition of $trans_k(\cdot;\cdot)$ properly for $k \geq 0$.

It is even easier to extend λ_{code} with pairs and references. For instance, in order to support references, all we need to assume is a type constructor $(\cdot)ref$ and the following functions of the given types, where **1** stands for the unit type.

$$\begin{array}{rcl} ref & : & \forall\alpha.\alpha \to (\alpha)ref \\ deref & : & \forall\alpha.(\alpha)ref \to \alpha \\ update & : & \forall\alpha.(\alpha)ref \to \alpha \to \mathbf{1} \end{array}$$

It is standard to assign dynamic semantics to *ref*, *deref* and *update*, and we omit the details. As an example, the following expression $`(\mathbf{lam}\, x.\mathbf{lam}\, y.update(y)(deref(x)))$ is translated into

$$\begin{array}{l} Lam(Lam(App(App(Lift(update),One), \\ \qquad App(Lift(deref),Shi(One))))) \end{array}$$

In [1], it is argued that a program corresponding to the following one would cause the problem of free variable evaluation to occur in MetaML [33] as *r* stores some open code when it is dereferenced.

```
let val r = ref `1
    val f = `(fn x => ^(r := `(x+1); `2)
in run (!r) end
```

[5] To avoid polymorphic recursion, we need to change `genAck(m-1)` into `Shi(Shi(genAck(m-1)))` in the implementation.

```
fun innerProd n = (* unstaged implementation of inner product *)
  let
      fun aux i v1 v2 sum =
        if i < n then aux (i+1) v1 v2 (sum + sub (v1, i) * sub (v2, i)) else sum
      withtype int -> int -> int array -> int array -> int
  in
      fn v1 => fn v2 => aux 0 v1 v2 0
  end
withtype int -> int array -> int array -> int

fun genInnerProd n = (* staged implementation of inner product *)
  let
      fun aux i v1 v2 sum =
        if i < n then
          aux (i+1) v1 v2 ``(^^sum + %(sub (^v1, i)) * sub (^^v2, i))
        else sum
      withtype {'g1,'g2}. int -> <'g1; int array> ->
                          <'g1,'g2; int array> -> <'g1,'g2; int> -> <'g1,'g2; int>
  in
     `(fn v1 => `(fn v2 => ^^(aux i `v1 ``v2 0)))
  end
withtype {'g1,'g2}. int -> <'g1; int array -> <'g2; int array -> int> >
```

Figure 8. A meta-programming example: inner product

This, however, cannot occur in λ^{+}_{code} as the above program is ill-typed: if r is assigned the type $(\langle \varepsilon, int \rangle)ref$, then it cannot be used to store open code; if r is assigned a type $(\langle G, int \rangle)ref$ for some nonempty G, then the code stored in it cannot be run.

With value restriction, it is also straightforward to support let-polymorphism in code: we can simply treat **let** $x = v$ **in** e **end** as syntactic sugar for $e[x \mapsto v]$.

What seems difficult is to treat pattern matching in code. One approach is to translate general pattern matching into the following form of fixed pattern matching for sum types,

$$\textbf{case } e_0 \textbf{ of } inl(x_1) \Rightarrow e_1 \mid inr(x_2) \Rightarrow e_2$$

and then introduce three code constructors *Inl*, *Inr* and *CaseOf* of the following types,

$$\forall\gamma\forall\alpha_1\forall\alpha_2.\langle\gamma,\alpha_1\rangle \rightarrow \langle\gamma,\alpha_1+\alpha_2\rangle$$
$$\forall\gamma\forall\alpha_1\forall\alpha_2.\langle\gamma,\alpha_2\rangle \rightarrow \langle\gamma,\alpha_1+\alpha_2\rangle$$
$$\forall\gamma\forall\alpha_1\forall\alpha_2\forall\alpha_3.$$
$$\langle\gamma,\alpha_1+\alpha_2\rangle \rightarrow \langle\alpha_1 :: \gamma,\alpha_3\rangle \rightarrow \langle\alpha_2 :: \gamma,\alpha_3\rangle \rightarrow \langle\gamma,\alpha_3\rangle$$

respectively. The rest becomes straightforward and we omit the details.

An alternative is to introduce a "*CaseOf*" code constructor for every declared datatype, and a code constructor for each value constructor associated with the datatype. This approach is more flexible than the previous one in practice but still has difficulty supporting nested patterns. We plan to introduce first-class patterns to better address the issue in the future.

6 Related Work and Conclusion

Meta-programming, which can offer a uniform and high-level view of the techniques for program generation, partial evaluation and run-time code generation, has been studied extensively in the literature.

An early reference to partial evaluation can be found in [13], where the three Futamura projections are presented for generating compilers from interpreters. The notion of *generating extensions*, which is now often called staged computation, is introduced in [12] and later expanded into multi-level staged computation [15, 14]. Most of the work along this line attempts to stage programs automatically (e.g., by performing binding-time analysis) and is done in an untyped setting.

In [7], a lambda-calculus $\lambda^{\Box}$ based on the intuitionistic modal logic S4 is presented for studying staged computation in a typed setting. Given a type A, a type constructor $\Box$, which corresponds to a modality operator in the logic S4, can be applied to A to form a type $\Box A$ for (closed) code of type A. With this feature, it becomes possible to verify whether a program with explicit staging annotations is indeed staged correctly. However, only closed code is allowed to be constructed in $\lambda^{\Box}$, and this can be a rigid restriction in practice. For instance, the usual power function, which is defined below,

```
fun power n x = (* it returns the nth power of x *)
  if n = 0 then 1 else x * power (n-1) x
```

can be staged in λ^{+}_{code} as follows in two different manners.

```
fun power1 n =
  if n = 0 then `(fn x => 1)
  else `(fn x => x * ^(power1 (n-1)) x)

fun power2 n = let
  fun aux i x =
    if i = 0 then `1 else `(^x * ^(aux (i-1) x))
in `(fn x => ^(aux n `x)) end
```

However, the second version (*power2*) does not have a counterpart in $\lambda^{\Box}$ as it involves the use of open code: there is a free variable in the code produced by *(aux n `x)*.

An approach to addressing the limitation is given in [6], where a type constructor $\bigcirc$ is introduced, which corresponds to the modality in discrete temporal logic for propositions that are true at the subsequent time moment. Given a type A, the type $\bigcirc A$ is for code, which may contain free variables, of type A.[6] This approach is es-

[6] Note that the function *run* is not present in $\lambda^{\bigcirc}$ for otherwise the problem of free variable evaluation would occur.

sentially used in the development of MetaML [33], an extension of ML that supports typed meta-programming by allowing the programmer to manually stage programs with explicit staging annotations. On one hand, when compared to untyped meta-programming in Scheme, the type system of MetaML offers an effective approach to capturing (pervasive) staging errors that occur during the construction of meta-programs. On the other hand, when compared to partial evaluation that performs automatic staging (e.g., in Similix), the explicit staging annotations in MetaML offer the programmer more flexibility and expressiveness.

However, as was first pointed out by Rowan Davies, the original type system of MetaML contained a defect caused by free variable evaluation (as the function *run* is available in MetaML) and there have since been a number of attempts to fix the defect. For instance, in [20], types for (potentially) open code are refined and it then becomes possible to form types for closed code only. In general, a value can be assigned a type for closed code only if the value does not depend on any free program variables. This approach is further extended [1] to handle references. Though sound, this approach also rules out code that is safe to run but does contain free program variables. We now use an example to illustrate this point. Let e_1 be the following expressions in λ^{+}_{code},

$$\textbf{lam}\ f.\text{`}(\textbf{lam}\ x.\hat{}(run(f(\text{`}x))))$$

and $e_2 = trans(e_1) = \textbf{lam}\ f.Lam(run(f(One)))$. Clearly, e_2 can be assigned a type of the following form:[7]

$$(\langle \tau_1 :: G_1, \tau_1 \rangle \rightarrow \langle \varepsilon, \langle \tau_2 :: G_2, \tau_3 \rangle \rangle) \rightarrow \langle G_2, \tau_2 \rightarrow \tau_3 \rangle$$

However, e_2 cannot be assigned a type in [20] or [1] as the type systems there cannot assign $f(\text{`}x)$ a "closed code" type. Though this is a highly contrived example, it nonetheless indicates some inadequacy in the notion of closed types captured by these type systems.

In [31], there is another type system that aims at assigning more accurate types to meta-programs in MetaML. In the type system, a notion of environment classifiers is introduced. Generally speaking, environment classifiers are used to explicitly name the stages of computation, and code is considered to be closed with respect to an environment classifier α if α can be abstracted. This approach is similar (at least in spirit) to the typing of *runST* in Haskell [16]. To some extent, an environment classifier resembles a type environment variable γ in λ^{+}_{code} and the type $(\alpha)\langle t \rangle^{\alpha}$ for code of type t that is closed with respect to an environment α relates to the type $\forall\gamma.\langle \gamma, t \rangle$ in λ^{+}_{code}.

Another approach to addressing the limitation of $\lambda^{\Box}$ is presented in [21]. Instead of refining the notion of (potentially open) code in $\lambda^{\bigcirc}$, the calculus $\nu^{\Box}$ in [21] relaxes the notion of closed code in $\lambda^{\Box}$ by extending $\lambda^{\Box}$ with a notion of *names* that is inspired by some developments in Nominal Logic [27] and FreshML [28]. Given an expression representing some code, the free variables in the code are represented as certain distinct names; the set of these names, which is called the *support* of the expression, is reflected in the type of the expression. The code represented by an expression can be executed only if the support of the expression is empty. Clearly, the notion of a support in $\nu^{\Box}$ corresponds to the notion of a type environment in λ_{code}. The primary difference between $\nu^{\Box}$ and λ_{code} as we see is that the development of the former is guided, implicitly or explicitly, by the notion of higher-order abstract syntax while the latter is based on a form of first-order abstract syntax.

[7] For example, this means e_2 can be applied to the function **lam** $x.Lift(x)$ (and the application evaluates to $Lam(One)$) but not to the function **lam** $x.x$.

There were certainly earlier attempts in forming typeful code representation. For instance, in a dependent type system such as LF, it is fairly straightforward to form a type $exp(t)$ in the *meta-language* for representing closed expressions of type t in the *object* language. Unfortunately, such typeful code representation seems unsuitable for meta-programming as the strict distinction between the meta-language and the object language makes it impossible for expressions in the meta-language to be handled in the object language. In particular, note that the code constructor *Lift* is no longer definable with this approach. An early approach to typeful code representation can be found in [25], where an inductively defined datatype is formed to support typeful representation for terms in the second-order polymorphic λ-calculus. This representation is higher-order and supports both reflection (i.e., to map the representation of an expression to the expression itself) and reification (i.e., to map an expression to the representation of the expression). However, it handles reification for complex values such as functions in a manner that seems too limited to support (practical) meta-programming. In [5], an approach is presented that implements (a form of) typeful h.o.a.s. in Haskell-like languages to represent simply typed λ-terms. With this approach, it is shown that an implementation of the normalizing function for simply typed λ-terms preserves types. However, the limitation of the approach is also severe: It does not support functions that take typeful h.o.a.s. as input (e.g., a function like *run* in λ_{code}).

In this paper, we present a novel approach to typed meta-programming that makes use of a form of first-order typeful code representation in which program variables are replaced with deBruijn indices. We form a language λ_{code} in which expressions representing code can be constructed through code constructors and then executed through a special function *run*. Although λ_{code} suffices to establish a theoretical foundation for meta-programming, it lacks proper syntax to support practical meta-programming. We address the issue by extending λ_{code} into λ^{+}_{code} with some meta-programming syntax adopted from Scheme and MetaML; we first form rules to directly type programs in λ^{+}_{code} and then define a translation from λ^{+}_{code} into λ_{code} for assigning dynamic semantics to λ^{+}_{code}. We also present examples in support of meta-programming with λ^{+}_{code}.

Furthermore, we feel that the concrete code representation in λ_{code} can be of great use in facilitating the understanding of meta-programming. For instance, the considerably subtle difference between $\langle \%\langle e \rangle \rangle$ and $\langle\langle \%e \rangle\rangle$ [31] can be readily explained in λ_{code}; the former and the latter are translated into $Lift(e')$ and $App(Lift(lift), e')$, respectively, where e' is the translation $trans(e)$ of e and *lift* is **lam** $x.Lift(x)$; $run(Lift(e'))$ reduces to e' and $run(App(Lift(lift), e'))$ reduces to $Lift(v')$, where v' is the value of $run(e')$ (assuming e' represents closed code); so the difference between $\langle \%\langle e \rangle \rangle$ and $\langle\langle \%e \rangle\rangle$ is clear: the former means e is not executed until the second stage while the latter, which requires e to be closed, indicates that e is executed at the first stage and its value is lifted into the second stage.

We also show that λ_{code} can be embedded into $\lambda_{2,G\mu}$ in a straightforward manner, establishing an intimate link between code constructors and guarded recursive datatypes. This embedding immediately gives rise to the possibility of constructing programs that perform analysis on code.

In future, we are interested in a more flexible approach to handling pattern matching in code. For this purpose, we seem to be in need of first-class patterns. Also, we plan to study how names, instead of deBruijn indices, can be used to form first-order typeful code representation, and we believe this is an important step towards implementing transformations for typed programs in a type-correct manner.

7 Acknowledgment

We thank Walid Taha for his valuable comments on a previous version of the paper and Joseph Hallett for his efforts on proofreading the paper. Also, we thank some anonymous referees, whose comments helped us raise the quality of the paper significantly.

8 References

[1] C. Calcagno, E. Moggi, and T. Sheard. Closed Types for a Safe Imperative MetaML. *Journal of Functional Programmng*, 2003. (to appear).

[2] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF. In *Proceedings of 16th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '89)*, pages 146–160, 1989.

[3] A. Church. A formulation of the simple type theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, 1982.

[5] O. Danvy and M. Rhiger. A Simple Take on Typed Abstract Syntax in Haskell-like Languages. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS '01)*, pages 343–358, Tokyo, Japan, March 2001.

[6] R. Davies. A temporal logic approach to binding-time analysis. In *Symposium on Logic in Computer Science (LICS '96)*, pages 184–195, 1996.

[7] R. Davies and F. Pfenning. A Modal Analysis of Staged Computation. *Journal of ACM*, 48(3):555–604, 2001.

[8] N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes mathematicae*, 34:381–392, 1972.

[9] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of Smalltalk-80 System. In *Proceedings of 11th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, 1984.

[10] S. Draves. Partial evaluation for media processing. *ACM Computing Surveys (CSUR)*, 30(3es):21, 1998.

[11] R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical Report #356, Computer Science Department, Indiana University, 1992.

[12] A. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.

[13] Y. Futamara. Partial evaluation of computation process. *Systems, computers, controls*, 2(5):45–50, 1971.

[14] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.

[15] N. D. Jones, P. Sestoft, and H. Søndergaard. An expriment in partial evaluation: the generation of a compiler generator. In *Rewriting Techniques and Applications*, pages 124–140. Springer-Verlag LNCS 202, 1985.

[16] J. Launchbury and S. Peyton-Jones. State in Haskell. *Lisp and Symbolic Computation*, pages 293–342, 1995.

[17] M. Leone and P. Lee. Optimizing ml with run-time code generation. In *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, PA, June 1996. ACM Press.

[18] H. Massalin. *An Efficient Implementation of Fundamental Operating System Services*. Ph. D. dissertation, Columbia University, 1992.

[19] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[20] E. Moggi, W. Taha, Z.-E.-A. Benaissa, and T. Sheard. An Idealized MetaML: Simpler, and More Expressiove. In *European Symposium on Programming (ESOP '99)*, pages 193–207. Springer-Verlag LNCS 1576, 1999.

[21] A. Nanevski and F. Pfenning. Meta-Programming with Names and Necessity. A previous version appeared in the *Proceedings of the International Conference on Functional Programming (ICFP 2002)*, pp. 206–217.

[22] S. Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available at `http://www.haskell.org/onlinereport/`, Feb. 1999.

[23] F. Pfenning. *Computation and Deduction*. Cambridge University Press. (to appear).

[24] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[25] F. Pfenning and P. Lee. A Language with Eval and Polymorphism. In *International Joint Conference on Theory and Practice in Software Development*, pages 345–359, Barcelona, Spain, March 1989. Springer-Verlag LNCS 352.

[26] R. Pike, B. Locanthi, and J. Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software – Practice and Experience*, 15(2):131–151, February 1985.

[27] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*. To appear. (A preliminary version appeared in the *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software* (TACS 2001), LNCS 2215, Springer-Verlag, 2001, pp 219–242.).

[28] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[29] T. Sheard, W. Taha, Z. Benaissa, and E. Pasalic. MetaML. Available at `http://www.cse.ogi.edu/PacSoft/projects/metaml/`.

[30] W. Taha, C. Calcagno, L. Huang, and X. Leroy. MetaOCaml. Available at `http://www.cs.rice.edu/~taha/MetaOCaml/`.

[31] W. Taha and M. F. Nielsen. Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 26–37, New Orleans, January 2003.

[32] W. Taha and T. Sheard. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997.

[33] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[34] A. Wright. Simple imperative polymorphism. *Journal of Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[35] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.