JFP **15** (6): 797–835, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796805005617 Printed in the United Kingdom

Meta-programming through typeful code representation

CHIYAN CHEN and HONGWEI XI
Computer Science Department, Boston University, Boston, MA, USA
(*e-mail:* {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 represented in terms of deBruijn indexes, and then formally demonstrate how such typeful code representation can be used to support meta-programming. With our approach, a particular interesting feature is that code becomes first-class values, which can be inspected as well as executed at run-time. The main contribution of the paper lies in the recognition and then the formalization of a novel approach to typed meta-programming that is practical, general and flexible.

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 many examples (kernel implementation (Massalin, 1992), graphics (Pike *et al.*, 1985), interactive media (Draves, 1998), method dispatch in object-oriented languages (Deutsch & Schiffman, 1984; Chambers & Ungar, 1989), etc.) where run-time code generation can be employed to reap significant gain in run-time performance (Leone & Lee, 1996). 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 non-empty 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,

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

where we make use of the backquote/comma notation in Scheme. 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 hygienic macros (Dybvig, 1992) 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.

[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.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

The first and foremost issue in typed meta-programming is the need for properly representing typed code. Intuitively, we need a type constructor $code(\cdot)$ such that for a given type τ, $code(\tau)$ 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 $code(\tau)$, $run(e)$ 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 $code(\tau)$ 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.) (Church, 1940; Pfenning & Elliott, 1988; Pfenning & Lee, 1989). For instance, the following declaration in Standard ML (SML) (Milner *et al.*, 1997) 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 (Xi *et al.*, 2003). For instance, we can declare a g.r. datatype constructor $HOAS(\cdot)$ and associate with it two value constructors *HOASlam* and *HOASapp* that are assigned the following types, respectively:[2]

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

Intuitively, for a given type τ, $HOAS(\tau)$ 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 $\lambda x : int.\lambda y : int \rightarrow int.y(x)$ is given below,

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

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

By associating with *HOAS* some extra value constructors, we can represent closed code of type τ as expressions of type $HOAS(\tau)$. In other words, we can define $code(\cdot)$ as $HOAS(\cdot)$. 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 (Xi *et al.*, 2003) 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

[2] Note that, unlike a similar inductively defined type constructor (Pfenning & Lee, 1989), *HOAS* cannot be inductively defined.

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

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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 that 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 in meta-programming, which we briefly explain as follows. Suppose we need to convert the following h.o.a.s. tree t, which is assigned the type $HOAS(HOAS(int) \rightarrow int)$, into some f.o.a.s. tree in order to run the code represented by t:

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

When making this conversion, we then need to apply the function *run* to a variable ranging over expressions of type $HOAS(HOAS(int))$. 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 (Taha & Sheard, 2000). This subtle problem, which seems rather technical, can lead to serious difficulties in establishing type soundness for typed meta-programming.

In this paper, we choose a form of first-order abstract syntax tree to represent code that not only supports direct open code manipulation but also avoids the subtle problem of free variable evaluation. As for the free program variables in open code, we use deBruijn indexes (de Bruijn, 1972) 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 a type environment, which is to be represented as a sequence of types; an expression of type $\langle G, \tau \rangle$ represents some code of type τ in which the free variables are assigned types according to G, and therefore the type for closed code of type τ is simply $\langle \epsilon, \tau \rangle$, where ϵ stands for 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 indexes 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.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

kinds	κ	::=	$type \mid env$
types	τ	::=	$a \mid \tau_1 \rightarrow \tau_2 \mid \langle G, \tau \rangle \mid \forall a : \kappa.\tau$
c-types	τ_c	::=	$\forall a_1 : \kappa_1 \ldots \forall a_m : \kappa_m.(\tau_1, \ldots, \tau_n) \Rightarrow \tau$
type env.	G	::=	$a \mid \epsilon \mid \tau :: G$
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	::=	$x \mid f \mid c(e_1, \ldots, e_n) \mid \mathbf{lam}\ x.e \mid e_1(e_2) \mid \mathbf{fix}\ f.e \mid \forall^+(v) \mid \forall^-(e)$
values	v	::=	$x \mid cc(v_1, \ldots, v_n) \mid \mathbf{lam}\ x.e \mid \forall^+(v)$
exp. var. ctx.	Γ	::=	$\emptyset \mid \Gamma, \underline{xf} : \tau$
typ. var. ctx.	Δ	::=	$\emptyset \mid \Delta, \underline{a} : \kappa$

Fig. 1. The syntax for λ_{code}.

The main contribution of the paper lies in the recognition and then the 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} with some syntax in section 3 to facilitate meta-programming. In section 4, we briefly mention an external language that is designed for the programmer to construct programs that can eventually be translated 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 and pattern matching. 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, for which we provide some explanation as follows:

- We use the kinds *type* and *env* for types and type environments, respectively, and a for variables ranging over types and type environments. In addition, we may use α and γ for the variables ranging over types and type environments, respectively.
- We use τ for types and G for type environments. A type environment assigns types to free expression variables in code. For instance, $bool :: int :: \epsilon$ is a type

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

environment which assigns the types *bool* and *int* to the first and the second expression variables, respectively. We use s 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 according to the type environment G. For instance, the expression $App(One, Shi(One))$ can be assigned the type $\langle (int \to int) :: int :: \epsilon, int \rangle$, which indicates 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 are assigned 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 indexes (de Bruijn, 1972), and the function *run* is for executing typed closed code represented by such expressions.
- We differentiate **lam**-variables x from **fix**-variables f; a **lam**-variable is a value but a **fix**-variable is not; only a value can be substituted for a **lam**-variable. We use $\underline{xf}$ for either a **lam**-variable or a **fix**-variable.
- We use $\forall^+(\cdot)$ and $\forall^-(\cdot)$ 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 $\forall^-(\forall^+(\mathbf{lam}\ x.x))$. Later, the presence of $\forall^+$ and $\forall^-$ 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 (Wright, 1995) by requiring that $\forall^+$ 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 that a variable can be declared at most once in an expression (type) variable context Γ (Δ). For an expression variable context Γ, we write $\mathbf{dom}(\Gamma)$ for the set of variables declared in Γ and $\Gamma(\underline{xf}) = \tau$ if $\underline{xf} : \tau$ is declared in Γ. A similar notation also applies to type variable contexts Δ.

We use a signature Σ to assign each constant c a *constant type* (or c-type, for short) of the following form,

$$\forall a_1 : \kappa_1 \ldots \forall a_m : \kappa_m.(\tau_1, \ldots, \tau_n) \Rightarrow \tau$$

where n indicates the arity of c. We write $c(e_1, \ldots, e_n)$ for applying a constant c of arity n to arguments $e_1, \ldots, e_n$. For a constant c of arity 0, we may write c for $c()$. The main reason for introducing constant types is to allow Proposition 2.1 to be stated as is now.

For convenience, we may write $\forall\Delta$ for a list of quantifiers $\forall a_1 : \kappa_1 \ldots \forall a_m : \kappa_m$, where $\Delta = \emptyset, a_1 : \kappa_1, \ldots, a_m : \kappa_m$. 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 (regular) type.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

$$
\begin{array}{rcl}
\mathit{Lift} & : & \forall\gamma.\forall\alpha.(\alpha) \Rightarrow \langle\gamma, \alpha\rangle \\
\mathit{Lam} & : & \forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle\alpha_1 :: \gamma, \alpha_2\rangle) \Rightarrow \langle\gamma, \alpha_1 \rightarrow \alpha_2\rangle \\
\mathit{App} & : & \forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle\gamma, \alpha_1 \rightarrow \alpha_2\rangle, \langle\gamma, \alpha_1\rangle) \Rightarrow \langle\gamma, \alpha_2\rangle \\
\mathit{Fix} & : & \forall\gamma.\forall\alpha.(\langle\alpha :: \gamma, \alpha\rangle) \Rightarrow \langle\gamma, \alpha\rangle \\
\mathit{One} & : & \forall\gamma.\forall\alpha.() \Rightarrow \langle\alpha :: \gamma, \alpha\rangle \\
\mathit{Shi} & : & \forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle\gamma, \alpha_1\rangle) \Rightarrow \langle\alpha_2 :: \gamma, \alpha_1\rangle \\
\mathit{run} & : & \forall\alpha.(\langle\epsilon, \alpha\rangle) \Rightarrow \alpha
\end{array}
$$

Fig. 2. The types of some constants in λ_{code}.

Kinding rules $\Delta \vdash s : \kappa$

$$\frac{\Delta(a) = \kappa}{\Delta \vdash a : \kappa}$$

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

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

$$\frac{\Delta, a : \kappa \vdash \tau : \mathit{type}}{\Delta \vdash \forall a : \kappa.\tau : \mathit{type}}$$

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

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

Fig. 3. The kinding rules for λ_{code}.

2.1 Static and dynamic semantics

We present the kinding rules for λ_{code} in Figure 3. We use a judgment of the form $\Delta \vdash \tau : \mathit{type}$ ($\Delta \vdash G : \mathit{env}$) to mean that τ (G) is a well-formed type (type environment) under Δ. It is certainly possible to use only the kind *type* in the formulation of λ_{code}. For instance, we may replace $\mathit{env}, \epsilon, ::$ with type, $\mathbf{1}$, $*$, respectively, where $\mathbf{1}$ is the unit type and $*$ is the type constructor for forming product types. Examples in which types are used to encode values (e.g. natural numbers) are particularly abundant in Haskell and are often referred to as *pearls* in functional programming. For instance, some of such examples can be found in (Hinze, 2001; Chen *et al.*, 2004b). However, we feel that the use of *env* allows for a cleaner presentation of λ_{code}. We use Θ for finite mappings defined below and $\mathbf{dom}(\Theta)$ for the domain of Θ.

$$\Theta \quad ::= \quad [] \mid \Theta[a \mapsto s]$$

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

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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

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

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

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

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

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

$$\frac{\Delta, a : \kappa; \Gamma \vdash v : \tau \quad \Delta \vdash \Gamma\ [\mathrm{ok}]}{\Delta; \Gamma \vdash \forall^+(v) : \forall a : \kappa.\tau}\ \textbf{(ty-Lam+)}$$

$$\frac{\Delta; \Gamma \vdash e : \forall a : \kappa.\tau \quad \Delta \vdash s : \kappa}{\Delta; \Gamma \vdash \forall^-(e) : \tau[a \mapsto s]}\ \textbf{(ty-Lam−)}$$

Fig. 4. The typing rules for λ_{code}.

Given a type variable context Δ and an expression variable context Γ, we write $\Delta \vdash \Gamma\ [\mathrm{ok}]$ 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 that $\Delta \vdash \Gamma\ [\mathrm{ok}]$ hold.

The typing rules for λ_{code} are listed in Figure 4. In the rule **(ty-Lam+)**, which introduces $\forall^+$, the premise $\Delta \vdash \Gamma\ [\mathrm{ok}]$ ensures that there are no free occurrences of a 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 $\tau = \forall a : \kappa.\tau_1$, then v is of the form $\forall^+(v_1)$.

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

Note that the proposition would not hold if we assigned constants (regular) types instead of c-types (while also allowing them to be values). For instance, One would then be a value of type $\forall\gamma.\forall\alpha.\langle \alpha :: \gamma, \alpha \rangle$, but One certainly is not of the form $\forall^+(v)$ for any v.

We use θ for finite mappings defined below:

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

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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

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

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

We assign dynamic semantics to λ_{code} through the use of evaluation contexts, which are defined as follows:

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

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 $\underline{xfs}$ for a sequence of *distinct* expression variables $\underline{xf}$:

$$\begin{array}{rcll}
comp(\underline{xfs}; Lift(v)) & = & v & \\
comp(\underline{xfs}, \underline{xf}; One) & = & \underline{xf} & \\
comp(\underline{xfs}, \underline{xf}; Shi(v)) & = & comp(\underline{xfs}; v) & \\
comp(\underline{xfs}; Lam(v)) & = & \mathbf{lam}\ x.comp(\underline{xfs}, x; v) & (\text{if } x \text{ is not in } \underline{xfs}) \\
comp(\underline{xfs}; App(v_1, v_2)) & = & (comp(\underline{xfs}; v_1))(comp(\underline{xfs}; v_2)) & \\
comp(\underline{xfs}; Fix(v)) & = & \mathbf{fix}\ f.comp(\underline{xfs}, f; v) & (\text{if } f \text{ is not in } \underline{xfs})
\end{array}$$

Note that *comp* is a function at meta-level. Intuitively, when applied to a sequence of distinct expression variables $\underline{xfs}$ and a value v representing some code, *comp* returns the code. For instance, we have:

$$\begin{array}{rcl}
comp(\cdot, x, f; App(One, Shi(One))) & = & f(x) \\
comp(\cdot; Lam(Lam(App(One, Shi(One))))) & = & \mathbf{lam}\ x_1.\mathbf{lam}\ x_2.x_2(x_1)
\end{array}$$

In practice, *comp* should probably be implemented as a function that compiles code representation at run-time into some form of machine code and then properly load the generated machine code into the run-time system.

Definition 2.2

We define redexes and their reductions as follows.

- $(\mathbf{lam}\ x.e)(v)$ is a redex, and its reduction is $e[x \mapsto v]$.
- $\mathbf{fix}\ f.e$ is a redex, and its reduction is $e[f \mapsto \mathbf{fix}\ f.e]$.
- $\forall^-(\forall^+(v))$ is a redex, and its reduction is v.
- $run(v)$ is a redex if $comp(\cdot; v)$ is defined, and its reduction is $comp(\cdot; v)$.

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

In the following presentation, we may write $\mathscr{D} :: \Delta; \Gamma \vdash e : \tau$ to mean that $\mathscr{D}$ is a typing derivation of $\Delta; \Gamma \vdash e : \tau$. We first state the following weakening lemma.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

Lemma 2.3
(Weakening) We have the following.

1. Assume that $\Delta, a : \kappa$ is a valid type variable context. If $\Delta; \Gamma \vdash e : \tau$ is derivable, then $\Delta, a : \kappa; \Gamma \vdash e : \tau$ is also derivable.
2. Assume that $\Delta \vdash \Gamma, \underline{xf} : \tau_0$ [ok] holds. If $\Delta; \Gamma \vdash e : \tau$ is derivable, then $\Delta; \Gamma, \underline{xf} : \tau_0 \vdash e : \tau$ [ok] is also derivable.

The proof of the weakening lemma is straightforward. We now state the substitution lemma as follows:

Lemma 2.4
(Substitution) We have the following:

1. Assume that $\Delta \vdash \Theta : \Delta_0$ holds and $\Delta, \Delta_0 \vdash s : \kappa$ is derivable. Then $\Delta \vdash s[\Theta] : \kappa$ is also derivable.
2. Assume that $\Delta; \Gamma \vdash (\Theta; \theta) : (\Delta_0; \Gamma_0)$ holds and $\Delta, \Delta_0; \Gamma, \Gamma_0 \vdash e : \tau$ is derivable. Then $\Delta; \Gamma \vdash e[\theta] : \tau[\Theta]$ is also derivable.

Proof
The proofs for (1) and (2) are completely standard, and follow from structural induction on the kinding derivation of $\Delta, \Delta_0 \vdash s : \kappa$ and the typing derivation of $\Delta, \Delta_0; \Gamma, \Gamma_0 \vdash e : \tau$, respectively. □

Given a value v of type $\langle \epsilon, \tau \rangle$, we clearly expect that $comp(v)$ is a closed expression of type τ. The following lemma establishes a generalized version of this statement, where the function $env(\cdot)$ is defined as follows that maps a given expression variable context to a type environment:

$$env(\emptyset) = \epsilon \qquad env(\Gamma, \underline{xf} : \tau) = \tau :: env(\Gamma)$$

Lemma 2.5
Let Γ be $\emptyset, \underline{xf}_1 : \tau_1, \ldots, \underline{xf}_n : \tau_n$, where $n \geqslant 0$. If $\emptyset \vdash \Gamma$ [ok] holds and $\emptyset; \emptyset \vdash v : \langle env(\Gamma), \tau \rangle$ is derivable, then $comp(\underline{xf}_1, \ldots, \underline{xf}_n; v)$ is defined and the typing judgment $\emptyset; \Gamma \vdash comp(\underline{xf}_1, \ldots, \underline{xf}_n; v) : \tau$ is derivable.

Proof
Let us use $\underline{xfs}$ for the sequence of variables $\underline{xf}_1, \ldots, \underline{xf}_n$. By Proposition 2.1, we know that v must be of one of the following forms: $Lift(v_1)$, One, $Shi(v_1)$, $Lam(v_1)$, $App(v_1, v_2)$ and $Fix(v_1)$. The proof now proceeds by structural induction on v.

- $v = Lift(v_1)$ for some value v_1. Then according to the c-type assigned to $Lift$, we know that $\emptyset; \emptyset \vdash v_1 : \tau$ is derivable. By Lemma 2.3, we know that $\emptyset; \Gamma \vdash v_1 : \tau$ is derivable. Note that $comp(\underline{xfs}; v) = v_1$, and we are done.
- $v = One$. Then $comp(\underline{xfs}; v) = \underline{xf}_n$. Clearly, $env(\Gamma) = \tau_n :: \ldots :: \tau_1 :: \epsilon$. Since $\emptyset; \emptyset \vdash v : \langle env(\Gamma), \tau \rangle$ is derivable, we have $\tau = \tau_n$. Therefore, we know that $\emptyset; \Gamma \vdash comp(\underline{xfs}; v) : \tau$ is derivable.
- $v = Shi(v_1)$ for some value v_1. Clearly, Γ is of the form $\Gamma_1, \underline{xf}_n$, where $\Gamma_1 = \emptyset, \underline{xf}_1 : \tau_1, \ldots, \underline{xf}_{n-1} : \tau_{n-1}$. Hence, $comp(\underline{xfs}; v) = comp(\underline{xfs}_1; v_1)$, where $\underline{xfs}_1 = \underline{xf}_1, \ldots, \underline{xf}_{n-1}$. According to the c-type assigned to Shi, we know that $\emptyset; \emptyset \vdash v_1 : \langle env(\Gamma_1), \tau \rangle$ is derivable. By induction hypothesis on $v_1, \emptyset; \Gamma_1 \vdash comp(\underline{xfs}_1; v_1) : \tau$ is derivable. By Lemma 2.3, $\emptyset; \Gamma \vdash comp(\underline{xfs}; v) : \tau$ is also derivable.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

- $v = \mathit{Lam}(v_1)$ for some value v_1. According to the c-type assigned to *Lam*, we know $\tau = \tau' \to \tau''$ for some types τ' and τ'', and $\emptyset;\emptyset \vdash v_1 : \langle \tau' :: \mathit{env}(\Gamma), \tau'' \rangle$ is derivable. Note that $\mathit{comp}(\underline{xfs}; v) = \mathbf{lam}\ x.\mathit{comp}(\underline{xfs}, x; v_1)$, where we assume that x does not occur in $\underline{xfs}$. Then by induction hypothesis on v_1, we have that $\emptyset;\Gamma, x : \tau' \vdash \mathit{comp}(\underline{xfs}, x; v_1) : \tau''$ is derivable. Hence, $\emptyset;\Gamma \vdash \mathbf{lam}\ x.\mathit{comp}(\underline{xfs}, x; v_1) : \tau' \to \tau''$ is derivable. Note that $\tau = \tau' \to \tau''$ and $\mathit{comp}(\underline{xfs}; v) = \mathbf{lam}\ x.\mathit{comp}(\underline{xfs}, x; v_1)$, and we are done.
- $v = \mathit{App}(v_1, v_2)$ for some values v_1 and v_2. According to the c-type assigned to *App*, we know that both $\emptyset;\emptyset \vdash v_1 : \langle \mathit{env}(\Gamma), \tau' \to \tau \rangle$ and $\emptyset;\emptyset \vdash v_2 : \langle \mathit{env}(\Gamma), \tau \rangle$ are derivable for some type τ'. By induction hypothesis, both $\emptyset;\Gamma \vdash \mathit{comp}(v_1) : \tau' \to \tau$ and $\emptyset;\Gamma \vdash \mathit{comp}(v_2) : \tau'$ are derivable. Hence, $\emptyset;\Gamma \vdash \mathit{comp}(v) : \tau$ is derivable since $\mathit{comp}(\underline{xfs}; v) = (\mathit{comp}(\underline{xfs}; v_1))(\mathit{comp}(\underline{xfs}; v_2))$.
- $v = \mathit{Fix}(v_1)$ for some value v_1. According to the c-type assigned to v_1, we know that $\emptyset;\emptyset \vdash v_1 : \langle \tau :: \mathit{env}(\Gamma), \tau \rangle$ is derivable. Note that $\mathit{comp}(\underline{xfs}; \mathit{Fix}(v_1)) = \mathbf{fix}\ f.\mathit{comp}(\underline{xfs}, f; v_1)$, where we assume that f does not occur in $\underline{xfs}$. By induction hypothesis on v_1, $\emptyset;\Gamma, f : \tau \vdash \mathit{comp}(\underline{xfs}, f; v_1) : \tau$ is derivable. Hence, $;\Gamma \vdash \mathit{comp}(\underline{xfs}; v)$ is also derivable.

We conclude the proof as all the cases are now completed. □

Theorem 2.6
(Subject Reduction) Assume $\emptyset;\emptyset \vdash e : \tau$ is derivable. If $e \to e'$ holds, then $\emptyset;\emptyset \vdash e' : \tau$ is derivable.

Proof
Let $\mathscr{D}$ be the typing derivation of $\emptyset;\emptyset \vdash e : \tau$. Assume $e = E[e_0]$ for some evaluation context E and redex e_0. The proof proceeds by structural induction on E. We present below the only interesting case where $E = []$, that is, $e = e_0$ is a redex. In this case, we analyze the structure of e as follows.

- $e = (\mathbf{lam}\ x.e_1)(v)$. Then the typing derivation $\mathscr{D}$ is of the following form:

$$\dfrac{\dfrac{\mathscr{D}' :: \emptyset;\emptyset, x : \tau' \vdash e_1 : \tau''}{\emptyset;\emptyset \vdash \mathbf{lam}\ x.e_1 : \tau' \to \tau''}\ \textbf{(ty-lam)} \qquad \emptyset;\emptyset \vdash v : \tau'}{\emptyset;\emptyset \vdash (\mathbf{lam}\ x.e_1)(v) : \tau''}\ \textbf{(ty-app)}$$

  where $\tau = \tau''$. By Lemma 2.4 (2), we know that $\emptyset;\emptyset \vdash e_1[x \mapsto v] : \tau''$ is derivable. Note that $e' = e_1[x \mapsto v]$, and we are done.
- $e = \mathbf{fix}\ f.e_1$. Then the typing derivation $\mathscr{D}$ is of the following form:

$$\dfrac{\mathscr{D}' :: \emptyset; f : \tau \vdash e_1 : \tau}{\emptyset;\emptyset \vdash \mathbf{fix}\ f.e_1 : \tau}\ \textbf{(ty-fix)}$$

  By Lemma 2.4 (2), we know that $\emptyset;\emptyset \vdash e_1[f \mapsto e] : \tau$ is derivable. Note that $e' = e_1[f \mapsto e]$, and we are done.
- $e = \forall^-(\forall^+(v))$. Then the typing derivation $\mathscr{D}$ is of the following form:

$$\dfrac{\dfrac{\mathscr{D}' :: \emptyset, a : \kappa;\emptyset \vdash v : \tau_0}{\emptyset;\emptyset \vdash \forall^+(v) : \forall a : \kappa.\tau_0}\ \textbf{(ty-Lam+)} \qquad \emptyset \vdash s : \kappa}{\emptyset;\emptyset \vdash \forall^-(\forall^+(v)) : \tau_0[a \mapsto s]}\ \textbf{(ty-Lam−)}$$

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

where $\tau = \tau_0[a \mapsto s]$. By Lemma 2.4 (2), we know that $\emptyset;\emptyset \vdash v : \tau$ is derivable. Note that $e' = v$, and we are done.

- $e = run(v)$ for some value v. According to the c-type assigned to *run*, we know that $\emptyset;\emptyset \vdash v : \langle \epsilon, \tau \rangle$ is derivable. Note that $e' = comp(\cdot; v)$. By Lemma 2.5, $\emptyset;\emptyset \vdash e' : \tau$ is derivable. Hence, we are done.

□

Lemma 2.7
Assume that $\emptyset;\emptyset \vdash e : \tau$ is derivable. Then either e is a value or $e = E[e_0]$ for some evaluation context E and redex e_0.

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

Theorem 2.8
(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
The theorem follows from Lemma 2.7 immediately. □

Combining Theorem 2.6 and Theorem 2.8, 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 implies that the problem of free variable evaluation can never occur in λ_{code}.

2.3 Meta-programming with λ_{code}

In theory, 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 Damas & Milner (1982)) 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)))
```

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

```
      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 $list(int) \to \langle int :: \epsilon, int\rangle$, 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 $int \to int$.

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}. This programming style is unwieldy if not completely impractical. It is analogous to writing meta-programs in Scheme without using the backquote/comma notation. Therefore, we are naturally motivated to provide some syntactic support so as to facilitate meta-programming through typeful code representation.

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

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.) datatypes (Xi *et al.*, 2003). 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 $FOAS(\cdot,\cdot)$, where $\mathrm{ML}_{2,G\mu}$ (Xi *et al.*, 2003) is an external language of $\lambda_{2,G\mu}$. The code constructors *Lift*, *One*, *Shi*, *App*, *Lam*, *Fix* have their counterparts *FOASlift*, *FOASone*, *FOASshi*, *FOASapp*, *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 $FOAS(\cdot,\cdot)$ represents $\langle\cdot,\cdot\rangle$. Formally, we define a translation $|\cdot|$ as follows,

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

which translates type environments and types in λ_{code} into types in $\lambda_{2,G\mu}$. The function *run* is implemented in Figure 5. We use `withtype` clauses for supplying type annotations. The type annotation for *run* indicates that *run* is expected to be assigned the type $\forall\alpha.FOAS(\mathbf{1},\alpha) \to \alpha$, which corresponds to the type $\forall\alpha.\langle\epsilon,\alpha\rangle \to \alpha$ in

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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

datatype ENV (type) =
  (unit) ENVnil
| {g:type,a:type} ENVcons (a * g) of a * ENV (g)

(* '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:type,a:type} FOAS (g,a) -> (ENV (g) -> a)

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

Fig. 5. Implementing code constructors and *run* with guarded recursive datatypes.

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 (built-in) function that can perform compilation at run-time and then upload the code generated from the compilation.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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 just such an example. Also, a typeful implementation of a continuation-passing style translation on simply typed programs can be found in Chen & Xi (2003).

In a study on typing dynamic typing (Baars & Swierstra, 2002), an example is given to demonstrate how a compilation function can be implemented in Haskell in a typeful manner that translates typeless first-order abstract syntax trees (representing simply typed terms) into values in Haskell. A key difference between this example and the one presented in Figure 5 lies in that the former uses typeless code representation while the latter uses typeful code representation. Recently, we have seen that the technique developed in Baars & Swierstra (2002) is subsequently employed in an application involving parsing combinators (Baars & Swierstra, 2004).

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

The expression variable context Γ is now defined as follows,

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

where $\underline{xf}@k$ stands for variables at level $k \geqslant 0$ and we use the name *staged variable* for $\underline{xf}@k$. 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 `(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 ^(e_0) is at level k; if an occurrence of **lam** $x.e_1$ or **fix** $f.e_1$ is at level k, then x or f is bound at level k. A declared staged variable $\underline{xf}@k$ in Γ simply indicates that $\underline{xf}$ is to be bound at level k.

3.1 Static semantics

It is rather involved to formulate rules for typing expressions in λ^{+}_{code}. We first introduce as follows some concepts that are needed in further development.

For each natural number k, let $\mathbf{pos}_k$ be the set $\{1,\ldots,k\}$, or formally $\{n \mid 0 < n \leqslant k\}$. We use $\mathscr{G}$ for a finite mapping associating a type environment to each n in $\mathbf{pos}_k$ for some k. In particular, we use $\emptyset$ for the mapping $\mathscr{G}$ such that $\mathbf{dom}(\mathscr{G}) = \mathbf{pos}_0$, which is the empty set. We write $\Delta \vdash^{\mathscr{G}}_{k} \Gamma$ [ok] to mean that

1. $\mathbf{dom}(\mathscr{G}) = \mathbf{pos}_k$, and
2. $\Delta \vdash \Gamma(\underline{xf}@n) : \mathit{type}$ for each $\underline{xf}@n \in \mathbf{dom}(\Gamma)$, where $n \leqslant k$ is required, and
3. $\Delta \vdash \mathscr{G}(n) : \mathit{env}$ for each $n \in \mathbf{dom}(\mathscr{G})$.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

In addition, we introduce the following definitions:

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

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

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

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

  for $k \in \mathbf{dom}(\mathscr{G})$, where $G = \mathscr{G}(k)$. We write $\mathscr{G}(\Gamma;\tau)$ for $\mathscr{G}(k;\Gamma;\tau)$ if $\mathbf{dom}(\mathscr{G}) = \mathbf{pos}_k$.
- Given $\mathscr{G}$ and G such that $\mathbf{dom}(\mathscr{G}) = \mathbf{pos}_k$, we use $\mathscr{G} + G$ for the mapping $\mathscr{G}_1$ such that $\mathbf{dom}(\mathscr{G}_1) = \mathbf{pos}_{k+1}$, $\mathscr{G}_1(n) = \mathscr{G}(n)$ for each $n \in \mathbf{pos}_k$ and $\mathscr{G}_1(k+1) = G$.

We now provide an example to facilitate the understanding of these concepts.

Example 3.1

Let $\mathscr{G} = \emptyset + \gamma_1 + \gamma_2 + \gamma_3$, that is, $\mathbf{dom}(\mathscr{G}) = \{1,2,3\}$ and $\mathscr{G}(i) = \gamma_i$ for $i = 1,2,3$. In addition, let $\Gamma = x_1^0@0 : \tau_1^0, x_1^1@1 : \tau_1^1, x_1^2@2 : \tau_1^2, x_2^2@2 : \tau_2^2, x_1^3@3 : \tau_1^3$. Then for any type τ, we have

$$\begin{array}{rcl} \mathscr{G}(1;\Gamma;\tau) & = & \langle \tau_1^1 :: \gamma_1, \tau\rangle \\ \mathscr{G}(2;\Gamma;\tau) & = & \langle \tau_1^1 :: \gamma_1, \langle \tau_2^2 :: \tau_2^1 :: \gamma_2, \tau\rangle\rangle \\ \mathscr{G}(3;\Gamma;\tau) & = & \langle \tau_1^1 :: \gamma_1, \tau, \langle \tau_2^2 :: \tau_2^1 :: \gamma_2, \langle \tau_1^3 :: \gamma_3, \tau\rangle\rangle\rangle \end{array}$$

In general, given $\mathscr{G}$ and Γ, we can compute a type environment $\mathscr{G}(k)(k;\Gamma)$ for each $k \in \mathbf{dom}(\mathscr{G})$. The need for these type environments is to be explained once the typing rules for λ^+_{code} are presented.

We use $\Delta;\Gamma \vdash^{\mathscr{G}}_k e : \tau$ for a typing judgment in λ^+_{code}, where we require that $\Delta \vdash^{\mathscr{G}}_k \Gamma$ [ok] hold. Intuitively, $\mathscr{G}(k)$ stands for the initial type environment for code at level k. We present the typing rules for λ^+_{code} in Figure 6, and we may write $\mathscr{D} :: \Delta;\Gamma \vdash^{\mathscr{G}}_k e : \tau$ to mean that $\mathscr{D}$ is a derivation of $\Delta;\Gamma \vdash^{\mathscr{G}}_k e : \tau$.

The rule **(ty-var-0)** means that a variable at level 0 can be referred to at any level $k \geqslant 0$, while the rule **(ty-var-1)** indicates that a variable at level $k > 0$ is available only at level k. In principle, we also have the option to make variables at level $k > 0$ available at succeeding levels. The reason for not adopting this option is briefly explained in section 3.3.

The rules **(ty-encode)** and **(ty-decode)** are the only ones that can change the level of a typing judgment in λ^+_{code}. Clearly, if an expression e can be assigned a type τ under $\Delta;\Gamma$, then the expression '(e) should be assigned a type of the form $\langle G,\tau\rangle$ under $\Delta;\Gamma$ for some type environment G. The challenging problem, however, is how such a type environment can be determined. One possible solution is to compute such a type environment in terms of Γ. While this is a simple solution, its limitation is rather severe. For instance, we could then only assign the expression '$(lamx.x)$ the type $\langle G, int \to int\rangle$ for the empty context $G = \epsilon$, though we should really be allowed

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

Typing rules $\Delta;\Gamma \vdash^{\mathscr{G}}_{k} e : \tau$

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

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

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

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

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

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

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

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

$$\frac{\Delta, a : \kappa;\Gamma \vdash^{\emptyset}_{0} v : \tau \quad \Delta \vdash^{\emptyset}_{0} \Gamma\ [\mathrm{ok}]}{\Delta;\Gamma \vdash^{\emptyset}_{0} \forall^{+}(v) : \forall a : \kappa.\tau}\ \textbf{(ty-Lam+)}$$

$$\frac{\Delta;\Gamma \vdash^{\emptyset}_{0} e : \forall a : \kappa.\tau \quad \Delta \vdash s : \kappa}{\Delta;\Gamma \vdash^{\emptyset}_{0} \forall^{-}(e) : \tau[a \mapsto s]}\ \textbf{(ty-Lam−)}$$

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

to assign `(**lam** $x.x$) the type $\langle G, int \to int\rangle$ for any G.[4] To remove the limitation, each typing judgment is associated with a mapping $\mathscr{G}$ that can be combined with Γ to compute the needed type environment G. Similarly, if an expression e can be assigned a type $\langle G, \tau\rangle$ under $\Delta;\Gamma$, then $\hat{}(e)$ should be given a type τ under $\Delta;\Gamma$. However, this can happen only if G is in a form required by the rule **(ty-decode)**. Note that there is a perfect symmetry between **(ty-encode)** and **(ty-decode)**. As an example, we present a complete typing derivation in Figure 7, where *id* is assumed to be some constant of c-type $\forall\alpha.(\alpha) \Rightarrow \alpha$.

Note that polymorphic code is only allowed to occur at level 0. To support polymorphism at level $k > 0$, that is, to allow polymorphic code at level $k > 0$, λ_{code} needs to be extended to support higher-order polymorphism (as is supported in the

[4] For instance, this is necessary if we need to assign a type to the following expression:

(**lam** *code*.`(**lam** $y$.$\hat{}$(*code*)($y$)))(`(**lam** $x.x$))

.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset,\gamma,\alpha;\emptyset,x@1:\alpha\vdash_1^{\emptyset+\gamma}:\alpha}\textbf{(ty-var-1)}}{\Sigma(id)=\forall\alpha.(\alpha)\Rightarrow\alpha\quad\emptyset,\gamma,\alpha;\emptyset,x@1:\alpha\vdash_0^{\emptyset}\text{`}(x):\langle\alpha::\gamma,\alpha\rangle}\textbf{(ty-encode)}}{\emptyset,\gamma,\alpha;\emptyset,x@1:\alpha\vdash_0^{\emptyset}id(\text{`}(x)):\langle\alpha::\gamma,\alpha\rangle}\textbf{(ty-cst)}}{\emptyset,\gamma,\alpha;\emptyset,x@1:\alpha\vdash_1^{\emptyset+\gamma}\hat{}(id(\text{`}(x))):\alpha}\textbf{(ty-decode)}}{\emptyset,\gamma,\alpha;\emptyset\vdash_1^{\emptyset+\gamma}\textbf{lam}\ x.\hat{}(id(\text{`}(x))):\alpha\to\alpha}\textbf{(ty-lam)}}{\emptyset,\gamma,\alpha;\emptyset\vdash_0^{\emptyset}\text{`}(\textbf{lam}\ x.\hat{}(id(\text{`}(x)))):\langle\gamma,\alpha\to\alpha\rangle}\textbf{(ty-encode)}}{\emptyset,\gamma;\emptyset\vdash_0^{\emptyset}\forall^+(\text{`}(\textbf{lam}\ x.\hat{}(id(\text{`}(x))))):\forall\alpha.\langle\gamma,\alpha\to\alpha\rangle}\textbf{(ty-Lam+)}}{\emptyset;\emptyset\vdash_0^{\emptyset}\forall^+(\forall^+(\text{`}(\textbf{lam}\ x.\hat{}(id(\text{`}(x)))))):\forall\gamma.\forall\alpha.\langle\gamma,\alpha\to\alpha\rangle}\textbf{(ty-Lam+)}$$

Fig. 7. A complete typing derivation in λ^{+}_{code}.

higher-order polymorphically typed λ-calculus λ_{ω}). For instance, the following two code constructors *Alli* and *Alle* are needed for constructing expressions representing polymorphic code:

$$\begin{aligned} Alli &: \forall\gamma.\forall a_1:type\to type.(\forall a_2:type.\langle\gamma,a_1(a_2)\rangle)\Rightarrow\langle\gamma,\forall a_2:type.a_1(a_2)\rangle\\ Alle &: \forall\gamma.\forall a_1:type\to type.\forall a_2:type.(\langle\gamma,\forall a_2:type.a_1(a_2)\rangle)\Rightarrow\langle\gamma,a_1(a_2)\rangle \end{aligned}$$

Given the complication involved, we are not to deal with such code constructors in this paper and postpone the issue of representing polymorphic code for future study.

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

We introduce some notations needed in the following presentation. For $n\geqslant 0$, we use $\forall^{+}_{n}(e)$ for $\forall^{+}(\ldots(\forall^{+}(e))\ldots)$, where there are n occurrences of $\forall^{+}$, and we use $\forall^{-}_{n}(e)$ similarly. Given an expression e, we write $Lift^{n}(e)$ for $Lift(\ldots(Lift(e))\ldots)$, where there are n occurrences of *Lift*. We define some functions in Figure 8, which basically generalize the code constructors we have. Given e, $e_1,\ldots,e_n$, we write $App^{n}(e)(e_1)\ldots(e_n)$ 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

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

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

$$\mathscr{G}(\Gamma;\tau)=\langle G_1(1;\Gamma),\ldots,G_k(k,\Gamma);\tau\rangle,$$

where we assume $\mathscr{G}=\emptyset+G_1+\ldots+G_k$.

We now use $\underline{xfs}$ for a sequence of staged variables, that is, $\underline{xfs}$ is of the form $\underline{xf}_1@k_1,\ldots,\underline{xf}_n@k_n$. We define $var_1(\underline{xfs};\underline{xf})$ as follows under the assumption that

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

$$
\begin{array}{rcl}
Lift_n & : & \forall\gamma_1 \ldots \forall\gamma_n.\forall\alpha.\alpha \to \langle \gamma_1, \ldots, \gamma_n; \alpha \rangle \\
Lift_n & = & \forall^+_{n+1}(\mathbf{lam}\ x.Lift^n(x)) \\
 & & \\
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 \\
Lam_1 & = & \forall^+_3(\mathbf{lam}\ x.Lam(x)) \\
Lam_{n+1} & = & \forall^+_{n+3}(\mathbf{lam}\ x.App(Lift(\forall^-_{n+2}(Lam_n)), x)) \\
 & & \\
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 \\
App_1 & = & \forall^+_3(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.App(x_1, x_2)) \\
App_{n+1} & = & \forall^+_{n+3}(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.App(App(Lift(\forall^-_{n+2}(App_n)), x_1), x_2)) \\
 & & \\
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 \\
Fix_1 & = & \forall^+_2(\mathbf{lam}\ x.Fix(x)) \\
Fix_{n+1} & = & \forall^+_{n+2}(\mathbf{lam}\ x.App(Lift(\forall^-_{n+1}(Fix_n)), x)) \\
 & & \\
One_n & : & \forall\gamma_1 \ldots \forall\gamma_n.\forall\alpha.\langle \gamma_1, \ldots, \gamma_{n-1}, \alpha :: \gamma_n; \alpha \rangle \\
One_1 & = & \forall^+_2(One) \\
One_{n+1} & = & \forall^+_{n+2}(Lift(\forall^-_{n+1}(One_n))) \\
 & & \\
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 \\
Shi_1 & = & \forall^+_3(\mathbf{lam}\ x.Shi(x)) \\
Shi_{n+1} & = & \forall^+_{n+3}(\mathbf{lam}\ x.App(Lift(\forall^-_{n+2}(Shi_n)), x))
\end{array}
$$

Fig. 8. The types of generalized code constructors.

$\underline{xf}@1$ occurs in $\underline{xfs}$:

$$
var_1(\underline{xfs}; \underline{xf}) = \begin{cases} One & \text{if } \underline{xfs} = \underline{xfs}_1, \underline{xf}@1; \\ Shi(var_1(\underline{xfs}_1; \underline{xf})) & \text{if } \underline{xfs} = \underline{xfs}_1, \underline{xf}_1@1 \text{ and } \underline{xf}_1 \neq \underline{xf}; \\ var_1(\underline{xfs}_1; \underline{xf}) & \text{if } \underline{xfs} = \underline{xfs}_1, \underline{xf}_1@k_1 \text{ and } k_1 \neq 1 \end{cases}
$$

For each $k > 1$, we define $var_k(\underline{xfs}; \underline{xf})$ as follows under the assumption that $\underline{xf}@k$ occurs in $\underline{xfs}$:

$$
var_k(\underline{xfs}; \underline{xf}) = \begin{cases} \forall^-_{k+1}(One_k) & \text{if } \underline{xfs} = \underline{xfs}_1, \underline{xf}@k; \\ \forall^-_{k+2}(Shi_k)(var_k(\underline{xfs}_1; \underline{xf})) & \text{if } \underline{xfs} = \underline{xfs}_1, \underline{xf}_1@k \text{ and } \underline{xf}_1 \neq \underline{xf}; \\ var_k(\underline{xfs}_1; \underline{xf}) & \text{if } \underline{xfs} = \underline{xfs}_1, \underline{xf}_1@k_1 \text{ and } k_1 \neq k \end{cases}
$$

Proposition 3.2

Assume that $\Delta \vdash^{\mathscr{G}}_k : \Gamma$ [ok] holds and $\Gamma = \underline{xf}_1@k_1 : \tau_1, \ldots, \underline{xf}_n@k_n : \tau_n$. Then the following typing judgment:

$$
\Delta; \emptyset \vdash var_{k_i}(\underline{xf}_1@k_1, \ldots, \underline{xf}_n@k_n; \underline{xf}_i) : \mathscr{G}(k_i; \Gamma; \tau_i)
$$

is derivable for each $1 \leqslant i \leqslant n$ such that $0 < k_i \leqslant k$ holds.

Proof

By a straightforward inspection of the related definitions. □

In Figure 9, we define a translation $trans_k(\cdot;\cdot)$ for each $k \geqslant 0$ that translates expressions in λ^+_{code} into ones in λ_{code}. A crucial property of $trans_k(\cdot;\cdot)$ is captured by the following lemma, which consists of the main technical contribution of the paper.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

$\boxed{trans_0(\cdot;\cdot)}$

$$
\begin{array}{rcl}
trans_0(\underline{xfs};\underline{xf}) & = & \underline{xf} \quad \text{if } \underline{xf}@0 \text{ occurs in } \underline{xfs} \\
trans_0(\underline{xfs};c(e_1,\ldots,e_n)) & = & c(trans_0(\underline{xfs};e_1),\ldots,trans_0(\underline{xfs};e_n)) \\
trans_0(\underline{xfs};\mathbf{lam}\ x.e) & = & \mathbf{lam}\ x.trans_0(\underline{xfs},x@0;e) \\
trans_0(\underline{xfs};e_1(e_2)) & = & trans_0(\underline{xfs};e_1)(trans_0(\underline{xfs};e_1)) \\
trans_0(\underline{xfs};\mathbf{fix}\ f.e) & = & \mathbf{fix}\ f.trans_0(\underline{xfs},f@0;e) \\
trans_0(\underline{xfs};\forall^+(e)) & = & \forall^+(trans_0(\underline{xfs};e)) \\
trans_0(\underline{xfs};\forall^-(e)) & = & \forall^-(trans_0(\underline{xfs};e)) \\
trans_0(\underline{xfs};\text{`}(e)) & = & trans_1(\underline{xfs};e)
\end{array}
$$

$\boxed{trans_1(\cdot;\cdot)}$

$$
\begin{array}{rcl}
trans_1(\underline{xfs};\underline{xf}) & = & Lift(\underline{xf}) \quad \text{if } \underline{xf}@0 \text{ occurs in } \underline{xfs} \\
trans_1(\underline{xfs};\underline{xf}) & = & var_1(\underline{xfs};\underline{xf}) \quad \text{if } \underline{xf}@1 \text{ occurs in } \underline{xfs} \\
trans_1(\underline{xfs};c(e_1,\ldots,e_n)) & = & \\
\multicolumn{3}{l}{App^n(Lift(\mathbf{lam}\ x_1\ldots\mathbf{lam}\ x_n.c(x_1,\ldots,x_n)))(trans_1(\underline{xfs};e_1))\ldots(trans_1(\underline{xfs};e_n))} \\
trans_1(\underline{xfs};\mathbf{lam}\ x.e) & = & Lam(trans_1(\underline{xfs},x@1;e)) \\
trans_1(\underline{xfs};e_1(e_2)) & = & App(trans_1(\underline{xfs};e_1),trans_1(\underline{xfs};e_2)) \\
trans_1(\underline{xfs};\mathbf{fix}\ f.e) & = & Fix(trans_1(\underline{xfs},f@1;e)) \\
trans_1(\underline{xfs};\text{`}(e)) & = & trans_2(\underline{xfs};e) \\
trans_1(\underline{xfs};\hat{}(e)) & = & trans_0(\underline{xfs};e)
\end{array}
$$

$\boxed{trans_k(\cdot;\cdot) \text{ for } k > 1}$

$$
\begin{array}{rcl}
trans_k(\underline{xfs};\underline{xf}) & = & Lift^k(\underline{xf}) \quad \text{if } \underline{xf}@0 \text{ occurs in } \underline{xfs} \\
trans_k(\underline{xfs};\underline{xf}) & = & var_k(\underline{xfs};\underline{xf}) \quad \text{if } \underline{xf}@k \text{ occurs in } \underline{xfs} \\
trans_k(\underline{xfs};c(e_1,\ldots,e_n)) & = & \\
\multicolumn{3}{l}{App_k^n(Lift^k(\mathbf{lam}\ x_1\ldots\mathbf{lam}\ x_n.c(x_1,\ldots,x_n)))(trans_k(\underline{xfs};e_1))\ldots(trans_k(\underline{xfs};e_n))} \\
trans_k(\underline{xfs};\mathbf{lam}\ x.e) & = & \forall_{k+2}^-(Lam_k)(trans_k(\underline{xfs},x@k;e)) \\
trans_k(\underline{xfs};e_1(e_2)) & = & \forall_{k+2}^-(App_k)(trans_k(\underline{xfs};e_1))(trans_k(\underline{xfs};e_2)) \\
trans_k(\underline{xfs};\mathbf{fix}\ f.e) & = & \forall_{k+1}^-(Fix_k)(trans_k(\underline{xfs},f@k;e)) \\
trans_k(\underline{xfs};\text{`}(e)) & = & trans_{k+1}(\underline{xfs};e) \\
trans_k(\underline{xfs};\hat{}(e)) & = & trans_{k-1}(\underline{xfs};e)
\end{array}
$$

Fig. 9. The definition of $trans_k(\cdot;\cdot)$ for $k \geqslant 0$.

Lemma 3.3

Assume that the typing judgment $\Delta;\Gamma \vdash_k^{\mathscr{G}} e : \tau$ is derivable in λ_{code}^+ and $\Gamma = \underline{xf_1}@k_1 : \tau_1,\ldots,\underline{xf_n}@k_n : \tau_n$. Then the following typing judgment:

$$\Delta;(\Gamma)_0 \vdash trans_k(\underline{xf_1}@k_1,\ldots,\underline{xf_n}@k_n;e) : \mathscr{G}(\Gamma;\tau)$$

is derivable in λ_{code}, where $(\Gamma)_0$ is defined as follows:

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

Proof

Let us use $\underline{xfs}$ to denote $\underline{xf_1}@k_1,\ldots,\underline{xf_n}@k_n$. The proof follows from structural induction on the derivation $\mathscr{D}$ of the typing judgment $\Delta;\Gamma \vdash_k^{\mathscr{G}} e : \tau$, and we have

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

the following cases:

- $\mathscr{D}$ is of the following form:

$$\frac{\Delta \vdash_k^{\mathscr{G}} \Gamma\ [\mathrm{ok}] \quad \Gamma(\underline{xf}@0) = \tau}{\Delta; \Gamma \vdash_k^{\mathscr{G}} \underline{xf} : \tau}\ \textbf{(ty-var-0)}$$

  where $e = \underline{xf}$. Given that $\Delta \vdash_k^{\mathscr{G}} \Gamma\ [\mathrm{ok}]$ holds, we have that $\Delta \vdash (\Gamma)_0\ [\mathrm{ok}]$ holds in λ_{code}. By the definition of $(\Gamma)_0$, we have $(\Gamma)_0(\underline{xf}) = \tau$. Therefore $\Delta; (\Gamma)_0 \vdash \underline{xf} : \tau$ is derivable. We have the following subcases.

  — $k = 0$. Then $trans_k(\underline{xfs}; e) = \underline{xf}$ and $\mathscr{G}(\Gamma; \tau) = \mathscr{G}(0; \Gamma; \tau) = \tau$. Clearly, $\Delta; (\Gamma)_0 \vdash trans_0(\underline{xfs}; e) : \mathscr{G}(\Gamma; \tau)$ is derivable in λ_{code}.

  — $k \geqslant 1$. Then $trans_k(\underline{xfs}; e) = Lift^k(\underline{xf})$ and

$$\mathscr{G}(\Gamma; \tau) = \mathscr{G}(k; \Gamma; \tau) = \langle G_1(1; \Gamma), \ldots, G_k(k; \Gamma); \tau \rangle,$$

  where we assume $\mathscr{G} = \emptyset + G_1 + \ldots + G_k$. According to the c-type assigned to *Lift*, $\Delta; (\Gamma)_0 \vdash trans_k(\underline{xfs}; e) : \mathscr{G}(\Gamma; \tau)$ is clearly derivable in λ_{code}.

- $\mathscr{D}$ is of the following form:

$$\frac{\Delta \vdash_k^{\mathscr{G}} \Gamma\ [\mathrm{ok}] \quad \Gamma(\underline{xf}@k) = \tau}{\Delta; \Gamma \vdash_k^{\mathscr{G}} \underline{xf} : \tau}\ \textbf{(ty-var-1)}$$

  where $e = \underline{xf}$. Note that $k > 0$ holds. By Proposition 3.2, we know that $\Delta; \emptyset \vdash var_k(\underline{xfs}; \underline{xf}) : \mathscr{G}(k; \Gamma; \tau)$ is derivable. Hence, $\Delta; (\Gamma)_0 \vdash var_k(\underline{xfs}, \underline{xf}) : \mathscr{G}(k; \Gamma; \tau)$ is also derivable by Lemma 2.3. Note that $trans_k(\underline{xfs}; e) = var_k(\underline{xfs}; \underline{xf})$, and we are done.

- $\mathscr{D}$ is of the following form:

$$\frac{\mathscr{D}' :: \Delta; \Gamma, x@k : \tau' \vdash_k^{\mathscr{G}} e' : \tau''}{\Delta; \Gamma \vdash_k^{\mathscr{G}} \textbf{lam}\ x.e' : \tau' \to \tau''}\ \textbf{(ty-lam)}$$

  where $e = \textbf{lam}\ x.e'$ and $\tau = \tau' \to \tau''$. We have the following subcases.

  — $k = 0$. By induction hypothesis on $\mathscr{D}'$, the following typing judgment

$$\Delta; (\Gamma)_0, x : \tau' \vdash trans_0(\underline{xfs}, x@0; e') : \mathscr{G}(\Gamma; \tau'')$$

  is derivable. Note that $\mathscr{G}(\Gamma; \tau'') = \mathscr{G}(0; \Gamma; \tau'') = \tau''$. Therefore, the following typing judgment

$$\Delta; (\Gamma)_0 \vdash \textbf{lam}\ x.trans_0(\underline{xfs}, x@0; e') : \tau' \to \tau''$$

  is derivable. Since $trans_0(\underline{xfs}; e) = \textbf{lam}\ x.trans_0(\underline{xfs}, x@0; e')$ and $\mathscr{G}(\Gamma; \tau) = \mathscr{G}(0; \Gamma; \tau) = \tau = \tau' \to \tau''$, we have that $\Delta; (\Gamma)_0 \vdash trans_0(\underline{xfs}; e) : \mathscr{G}(\Gamma; \tau)$ is derivable.

  — $k = 1$. By induction hypothesis on $\mathscr{D}'$, the following typing judgment

$$\Delta; (\Gamma)_0, x : \tau' \vdash trans_1(\underline{xfs}, x@1; e') : \mathscr{G}(\Gamma, x@1 : \tau'; \tau'')$$

  is derivable. Assume $\mathscr{G} = \epsilon + G$, and we have $\mathscr{G}(\Gamma, x@1 : \tau'; \tau'') = \mathscr{G}(1; \Gamma, x@1 : \tau'; \tau'') = \langle G(1; \Gamma, x@1 : \tau'), \tau \rangle = \langle \tau' :: G(1; \Gamma), \tau'' \rangle$. Hence,

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

according to the c-type assigned to *Lam*, the following typing judgment

$$\Delta;(\Gamma)_0 \vdash Lam(trans_1(\underline{xfs}, x@1; e')) : \langle G(1;\Gamma), \tau' \to \tau'' \rangle$$

is derivable. Note that $trans_1(\underline{xfs}; e) = Lam(trans_1(\underline{xfs}, x@1; e'))$ and

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

and we are done.

— $k > 1$. This subcase is similar to the previous one where $k = 1$, and we thus omit the details.

- $\mathcal{D}$ is of the following form:

$$\frac{\mathcal{D}' :: \Delta;\Gamma \vdash_k^{\mathcal{G}} e' : \tau' \to \tau'' \quad \mathcal{D}'' :: \Delta;\Gamma \vdash_k^{\mathcal{G}} e'' : \tau'}{\Delta;\Gamma \vdash_k^{\mathcal{G}} e'(e'') : \tau''} \textbf{ (ty-app)}$$

where $e = e'(e'')$ and $\tau = \tau''$. We have the following subcases.

— $k = 0$. By induction hypothesis on $\mathcal{D}'$, the following typing judgment

$$\Delta;(\Gamma)_0 \vdash trans_0(\underline{xfs}; e') : \mathcal{G}(\Gamma; \tau' \to \tau'')$$

is derivable. Also by induction hypothesis on $\mathcal{D}''$, the following typing judgment

$$\Delta;(\Gamma)_0 \vdash trans_0(\underline{xfs}; e'') : \mathcal{G}(\Gamma; \tau')$$

is derivable. Note that $trans_0(\underline{xfs}; e) = trans_0(\underline{xfs}; e')(trans_0(\underline{xfs}; e''))$, and $\mathcal{G}(\Gamma; \tau' \to \tau'') = \mathcal{G}(0;\Gamma; \tau' \to \tau'') = \tau' \to \tau''$, and $\mathcal{G}(\Gamma; \tau') = \mathcal{G}(0;\Gamma;\tau') = \tau'$, and $\mathcal{G}(\Gamma;\tau) = \mathcal{G}(0;\Gamma;\tau) = \tau = \tau''$. Hence, $\Delta;(\Gamma)_0 \vdash trans_0(\underline{xfs}; e) : \mathcal{G}(\Gamma;\tau)$ is also derivable.

— $k = 1$. By induction hypothesis on $\mathcal{D}'$, the following typing judgment

$$\Delta;(\Gamma)_0 \vdash trans_1(\underline{xfs}; e') : \mathcal{G}(\Gamma; \tau' \to \tau'')$$

is derivable. Also by induction hypothesis on $\mathcal{D}''$, the following typing judgment

$$\Delta;(\Gamma)_0 \vdash trans_1(\underline{xfs}; e'') : \mathcal{G}(\Gamma; \tau')$$

is derivable. Assume $\mathcal{G} = \epsilon + G$. Then $\mathcal{G}(\Gamma; \tau' \to \tau'') = \mathcal{G}(1;\Gamma;\tau' \to \tau'') = \langle G(1;\Gamma), \tau' \to \tau'' \rangle$ and $\mathcal{G}(\Gamma;\tau') = \mathcal{G}(1;\Gamma;\tau') = \langle G(1;\Gamma), \tau' \rangle$. According to the type assigned to *App*, we know that the following typing judgment

$$\Delta;(\Gamma)_0 \vdash App(trans_1(\underline{xfs}; e'), trans_1(\underline{xfs}; e'')) : \langle G(1;\Gamma), \tau'' \rangle$$

is derivable. Note that $trans_1(\underline{xfs}; e) = App(trans_1(\underline{xfs}; e'), trans_1(\underline{xfs}; e''))$ and $\mathcal{G}(\Gamma;\tau'') = \mathcal{G}(1;\Gamma;\tau'') = \langle G(1;\Gamma), \tau'' \rangle$, and we are done.

— $k > 1$. This subcase is similar to the previous one where $k = 1$.

- $\mathcal{D}$ is of the following form:

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

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

where $e = \text{`}(e')$ and $\tau = \langle G(k+1;\Gamma), \tau' \rangle$. By induction hypothesis on $\mathscr{D}'$, we can derive $\Delta;(\Gamma)_0 \vdash \mathit{trans}_{k+1}(\underline{xfs};\text{`}(e')) : \mathscr{G}'(\Gamma;\tau')$ in λ_{code}, where $\mathscr{G}' = \mathscr{G} + G$. Note that $\mathit{trans}_k(\underline{xfs};\text{`}(e')) = \mathit{trans}_{k+1}(\underline{xfs};e')$ and

$$\mathscr{G}'(\Gamma;\tau') = \mathscr{G}'(k+1;\Gamma;\tau') = \mathscr{G}(k;\Gamma;\langle G(k+1;\Gamma),\tau'\rangle) = \mathscr{G}(\Gamma;\langle G(k+1;\Gamma),\tau'\rangle).$$

Hence, $\Delta;(\Gamma)_0 \vdash \mathit{trans}_k(\underline{xfs};e) : \mathscr{G}(\Gamma;\tau)$ is derivable.

- $\mathscr{D}$ is of the following form:

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

where $\mathscr{G} = \mathscr{G}' + G$ and $e = \hat{}(e')$. Note that $k > 0$ holds in this case. By induction hypothesis on $\mathscr{D}'$, the following typing derivation

$$\Delta;(\Gamma)_0 \vdash \mathit{trans}_{k-1}(\underline{xfs};e') : \mathscr{G}'(\Gamma;\langle G(k;\Gamma),\tau\rangle)$$

is derivable. Note that we have $\mathit{trans}_{k+1}(\underline{xfs};\hat{}(e)) = \mathit{trans}_k(\underline{xfs};e)$ and $\mathscr{G}(\Gamma;\tau) = \mathscr{G}(k;\Gamma;\tau) = \mathscr{G}'(k-1;\Gamma;\langle G(k;\Gamma),\tau\rangle) = \mathscr{G}'(\Gamma;\langle G(k;\Gamma),\tau\rangle)$. Hence, the typing judgment $\Delta;(\Gamma)_0 \vdash \mathit{trans}_k(\underline{xfs};e) : \mathscr{G}(\Gamma;\tau)$ is derivable.

The other cases can be handled similarly. For instance, the case where the last rule applied in $\mathscr{D}$ is **(ty-cst)** is analogous to the one where the last rule applied in $\mathscr{D}$ is **(ty-app)**. We thus omit the further details. □

Given an expression e in λ^{+}_{code}, we write $\mathit{trans}(e)$ for $\mathit{trans}_0(\emptyset;e)$ (if it is well-defined) and call it the translation of e.

Theorem 3.4
Assume that $\emptyset;\emptyset \vdash^{\emptyset}_0 e : \tau$ is derivable. Then $\emptyset;\emptyset \vdash \mathit{trans}(e) : \tau$ is derivable.

Proof
This immediately follows from Lemma 3.3. □

The programmer can now write meta-programs in λ^{+}_{code} that make use of meta-programming syntax, and these programs can then be translated into λ_{code} automatically. In other words, we may just treat meta-programming syntax merely as a form of syntactic sugar. This is precisely the significance of Theorem 3.4.

We conclude this section with an example to show how the type system of λ^{+}_{code} prevents free variable evaluation. Let us recall the following example in MetaML,

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

whose evaluation leads to free variable evaluation. In λ^{+}_{code}, the example corresponds to $e = \text{`}(\textbf{lam}\ x.\hat{}(\mathit{run}(\text{`}(x))))$. Clearly,

$$\begin{aligned}
\mathit{trans}(e) &= \mathit{trans}_0(\emptyset;\text{`}(\textbf{lam}\ x.\hat{}(\mathit{run}(\text{`}(x))))) \\
&= \mathit{trans}_1(\emptyset;\textbf{lam}\ x.\hat{}(\mathit{run}(\text{`}(x)))) \\
&= \mathit{Lam}(\mathit{trans}_1(\emptyset, x@1;\hat{}(\mathit{run}(\text{`}(x))))) \\
&= \mathit{Lam}(\mathit{trans}_0(\emptyset, x@1;\mathit{run}(\text{`}(x)))) \\
&= \mathit{Lam}(\mathit{run}(\mathit{trans}_0(\emptyset, x@1;\text{`}(x)))) \\
&= \mathit{Lam}(\mathit{run}(\mathit{trans}_1(\emptyset, x@1;x))) \\
&= \mathit{Lam}(\mathit{run}(\mathit{One}))
\end{aligned}$$

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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 $\langle \epsilon, \tau \rangle$ for some τ. Therefore, *trans*(*e*) is ill-typed. By Theorem 3.4, *e* is also ill-typed in λ^{+}_{code} and is thus properly rejected.

To see that *e* is not typable from a different angle, let us try to assign a type to *e* in λ^{+}_{code} directly. If this is possible, then a derivation $\mathscr{D}_0$ of the following form can be constructed for some G_0, τ_1, τ_2:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots \quad \cdots}{\emptyset;\emptyset, x@1 : \tau_1 \vdash_0^{\emptyset} run(`(x)) : \langle \tau_1 :: G_0, \tau_2 \rangle}\ \textbf{(ty-cst)}}{\emptyset;\emptyset, x@1 : \tau_1 \vdash_1^{\emptyset+G_0} \hat{}(run(`(x))) : \tau_2}\ \textbf{(ty-decode)}}{\emptyset;\emptyset \vdash_1^{\emptyset+G_0} \textbf{lam}\ x.\hat{}(run(`(x))) : \tau_1 \to \tau_2}\ \textbf{(ty-lam)}}{\emptyset;\emptyset \vdash_0^{\emptyset} `(\textbf{lam}\ x.\hat{}(run(`(x)))) : \langle G_0, \tau_1 \to \tau_2 \rangle}\ \textbf{(ty-encode)}$$

Note that we can only assign $`(x)$ a type of the form $\langle \tau_1 :: G_1, \tau_1 \rangle$ for some G_1:

$$\dfrac{\emptyset;\emptyset, x@1 : \tau_1 \vdash_1^{\emptyset+G_1} x : \tau_1}{\emptyset;\emptyset, x@1 : \tau_1 \vdash_0^{\emptyset} `(x) : \langle \tau_1 :: G_1, \tau_1 \rangle}\ \textbf{(ty-encode)}$$

In other words, $`(x)$ cannot be given a type for closed code. Hence, *run*$(`(x))$ is not typable, and the derivation $\mathscr{D}_0$ is impossible to construct.

3.3 Some remarks

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

No Free Named Variables at Level $k > 0$ In λ^{+}_{code}, no free named variables are allowed at any level $k > 0$. For instance, the expression $`(x + y)$ is considered illegal as it contains free named variables $x$ and $y$ at level 1, which can not be properly handled by the translation defined in Figure 9. On the other hand, the expression $`(\textbf{lam}\ x.\textbf{lam}\ y.x + y)$ is legal, where x and y are bound named variables at level 1. This expression translates into the following expression in λ_{code}:

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

Disallowing free named variable at any level $k > 0$ may cause certain inconvenience in programming. For instance, the following code, where the syntax should be able to be easily related to that of λ^{+}_{code} by someone familiar with ML, involves the use of free variable names at level 1:

```
let val code = '(x + y) in
   ( '(fn x => fn y => ^code), '(fn y => fn x => ^code) )
end
```

One possible solution here is to rewrite the above code as follows:

```
let fun code (x, y) = '(^x + ^y) in
   ( '(fn x => fn y => ^(code ('x, 'y))),
     '(fn y => fn x => ^(code ('x, 'y))) )
end
```

thus obviating the need for free variable names at level 1.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

In order to support free named variables at level $k > 0$, it seems that a notion of names is needed to be directly incorporated into the type system of the underlying language, which we think is highly nontrival. We refer the interested reader to Nanevski & Pfenning (n.d.) for some related work in this direction. We have here made a design choice to choose the use of nameless variables in code representation. This choice is partly influenced by our experience with MetaML (Taha & Sheard, 2000), which does not support free named variables at level $k > 0$, either.

Insertion of Explicit Shifts Suppose that e represents some closed code of type τ. If e is given the type $\langle \epsilon, \tau \rangle$, then the following expression cannot be assigned any type in λ^{+}_{code}: ‘(**lam** x.^(e)) as it requires that e be some code containing at least one free variable. To fix the problem, the code constructor *Shi* needs to be inserted explicitly: ‘(**lam** x.^(*Shi*(e))). However, if e is instead given the type $\forall\gamma.\langle \gamma, \tau \rangle$, which is more likely than not, then such an insertion becomes unnecessary as γ can be instantiated properly to represent a context containing at least one variable. For instance, Example 2, which is to be presented in section 4, sheds some light on this issue.

Mixed Use of Named and Nameless Variables The translation defined in Figure 9 indicates that named variables at level $k > 0$ are really just nameless variables, that is, deBruijn indexes in disguise. For instance, in the expression ‘(**lam** x.**lam** $y.x + y$), x and y are really just the second (*Shi*(*One*)) and the first (*One*) deBruijn indexes. As a matter of fact, this expression can be written in various equivalent forms such as:

- ‘(**lam** x.**lam** y.^(*Shi* *One*) + ^(*One*))
- ‘(**lam** x.**lam** $y.x$ + ^(*One*))
- ‘(**lam** x.**lam** y.^(*Shi* *One*) + y)
- ‘(**lam** x.**lam** y.^(*Shi* ‘(y)) + y)

However, the use of such forms should probably be discouraged unless some specific reasons are present.

Polymorphic Code As indicated by the typing rules **(ty-Lam+)** and **(ty-Lam-)**, polymorphism is only allowed at level 0 in λ^{+}_{code}. In other words, polymorphic code cannot be constructed in λ^{+}_{code}. For example, the code ‘($\forall^{+}$(**lam** $x.x$)) cannot be assigned any type in λ^{+}_{code}. On the other hand, the code $\forall^{+}$‘(**lam** $x.x$) can be given the type $\forall\gamma.\forall\alpha.\langle \gamma, \alpha \rightarrow \alpha \rangle$. However, there is no fundamental limitation on supporting polymorphic code. For instance, with the introduction of higher-order polymorphism into λ^{+}_{code}, polymorphic code can be readily supported (Chen *et al.*, 2004a). However, higher-order polymorphism can result in many complications in programming language design and implementation. As is supported by the case of MetaML (Taha & Sheard, 2000), we feel that meta-programming with no direct support for polymorphic code is still a viable practice. As to whether there are realistic cases where polymorphic code is truly needed, the answer is positive. Please see Chen *et al.* (2004a) for some distributed meta-programming examples involving sophisticated use of polymorphic code.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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 $`(\textbf{lam}\ x.\hat{}(f\ `x))$ and $e' = trans(e) = Lam(f(One))$.

- Let f be the identity function. Then e' evaluates to $Lam(One)$, which represents the code for the identity function $\textbf{lam}\ y.y$.
- Let f be the shift function $\textbf{lam}\ y.Shi(y)$. Then e' evaluates to $Lam(Shi(One))$, which represents the code $\textbf{lam}\ y.z$ for some free variable z that is distinct from y; there is no binding between Lam and One in e'.
- Let f be the lift function $\textbf{lam}\ y.Lift(y)$. Then e' evaluates to $Lam(Lift(One))$ and $run(e')$ evaluates to the function $\textbf{lam}\ y.One$ (not to $\textbf{lam}\ y.Lift(y)$); there is simply 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 $\langle \epsilon, \tau \rangle$.[5]

Cross-Stage Persistence In meta-programming, a situation often arises where a value defined at a previous level needs to be used at a following level. For instance, in the expression $`(\textbf{lam}\ x.x + x)$, the function $+$, which is defined at level 0, is used at level 1. This is called cross-stage persistence (CSP) (Taha & Sheard, 1997). As is indicated in the typing rules **(ty-var-0)** and **(ty-cst)**, CSP for both variables and constants at level 0 is implicit in λ^{+}_{code}. However, for variables introduced at level $k > 0$, CSP needs to be explicit. For instance, $`(\textbf{lam}\ x.`(\textbf{lam}\ y.y(x)))$ is ill-typed in λ^{+}_{code} as the variable x is introduced at level 1 but used at level 2. To make it typable, the programmer needs to insert % in front of x: $`(\textbf{lam}\ x.`(\textbf{lam}\ y.y(\%x)))$, 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 $`\%$, that is, $Lift(e)$ can be treated as $`(\%e)$ for any expression e. We present some further explanation on the issue of lifting variables across levels.

Cross-Level Variable Lifting The following rule **(ty-var-0)** indicates that at any level $k \geqslant 0$ we can refer to a variable at level 0:

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

Therefore, a natural question is whether at level k we can refer to a variable at level k_0 for any $k_0 \leqslant k$? In other words, can we modify the rule **(ty-var-0)** to the following one?

$$\frac{\Delta \vdash^{\mathscr{G}}_{k} \Gamma\ [\mathrm{ok}] \quad \Gamma(\underline{xf}@k_0) = \tau \ \text{ for some } k_0 \leqslant k}{\Delta; \Gamma \vdash^{\mathscr{G}}_{k} \underline{xf} : \tau}\ \textbf{(ty-var-0')}$$

[5] In $\nu^{\Box}$ (Nanevski & Pfenning, n.d.), e_0 cannot be typed, either. However, e_0 can be typed in the current implementation of MetaML (Sheard *et al.*, n.d.) and MetaOCaml (Taha *et al.*, n.d.); in the former e_0 evaluates to 1 but in the latter the evaluation of e_0 raises a run-time exception caused by free variable evaluation.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

The answer is yes. However, we need to modify the function $trans_k(\cdot;\cdot)$ as well for the case $k > 1$ in order to accommodate the rule **(ty-var-0')**:

$$trans_k(\underline{xfs};\underline{xf}) \quad = \quad Lift_{k_0,k}(var_{k_0}(\underline{xfs};\underline{xf})) \quad \text{if } \underline{xf}\,@k_0 \text{ occurs in } \underline{xfs}$$

where $Lift_{k_0,k}$ is a function that can be defined as follows:

$$\begin{array}{lcl} Lift_{k_0,k} & : & \forall\gamma_1 \ldots \forall\gamma_k.\forall\alpha.\langle\gamma_1,\ldots,\gamma_{k_0};\alpha\rangle \rightarrow \langle\gamma_1,\ldots,\gamma_k;\alpha\rangle \\ Lift_{k_0,k} & = & \forall^+_{k+1}(\mathbf{lam}\ x.\forall^-_{k_0+2}(App_{k_0})(Lift^{k_0}(Lift_{k-k_0}))(x)) \end{array}$$

Theorem 3.4 can then be proven again. Notice that the rule **(ty-var-1)** is no longer needed in the presence of the rule **(ty-var-0')**. While this rule makes it automatic to lift variables across levels, it also seems to make it difficult to detect staging errors in practice. Therefore, for some practical reasons we only allow variables at level 0 to be lifted across levels automatically in our design. To lift a variable x at level $k > 0$ to the next level, the programmer needs to write $\%x$ instead; the symbol $\%$ can be used repeatedly if a variable needs to be lifted across several levels, but such a situation seems rare at least.

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.

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

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 Damas & Milner (1982) that supports the usual let-polymorphism. Like in Haskell (Peyton Jones *et al.*, 1999), if the type of a recursive function is given, then polymorphic recursion is allowed in the definition of the function.

4.1 Some meta-programming examples

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

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

```
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>
```

Fig. 10. A staged implementation of the Ackerman function.

Example 1

The previously defined function *genEvalPoly* can now be implemented as follows, using no explicit code constructors.

```
fun genEvalPoly (p) =
  let
     fun aux p x =
       if null (p) then ‘(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. Also, note a use of the CSP operator % in this example.

Example 2

The following program implements the Ackerman 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* in Figure 10 such that the function returns the code for computing *ack*(*m*) when applied to a given natural number *m*. We use the syntax `‘(fun f (n) => ...)` for

```
‘(fix f => (fn n => ...))’,
```

which translates into something of the form *Fix*(*Lam*(...)). This example shows an interesting use of recursion at level 1. Also, we point out that polymorphic recursion is required in this example.[6]

[6] To avoid polymorphic recursion, we need to change `genAck(m-1)` into `Shi(Shi(genAck(m-1)))` in the implementation. However, the dynamic semantics of the program is not affected by the change.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

```
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> >
```

Fig. 11. A meta-programming example: inner product.

Example 3

We contrast an unstaged implementation of inner product (*innerProd*) with a staged implementation of inner product (*genInnerProd*) in Figure 11. 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 + %(sub (v1, 0)) * sub (v2, 0) + %(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 a use of the CSP operator %.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

5 Language extensions

It is straightforward to extend λ_{code} (and subsequently λ^{+}_{code}) to support additional language features such as conditionals, pairs, references, etc., and we present a brief outline as follows.

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 c-type:

$$\forall\gamma.\forall\alpha.(\langle\gamma, bool\rangle, \langle\gamma, \alpha\rangle, \langle\gamma, \alpha\rangle) \Rightarrow \langle\gamma, \alpha\rangle$$

We then define If_n as follows for $n \geqslant 1$:

$$\begin{aligned}
If_n &: \forall\gamma_1 \ldots \forall\gamma_n.\forall\alpha. \\
&\quad \langle\gamma_1, \ldots, \gamma_n; bool\rangle \rightarrow \langle\gamma_1, \ldots, \gamma_n; \alpha\rangle \rightarrow \langle\gamma_1, \ldots, \gamma_n; \alpha\rangle \rightarrow \langle\gamma_1, \ldots, \gamma_n; \alpha\rangle \\
If_1 &= \forall^{+}_{2}(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.\mathbf{lam}\ x_3.If(x_1, x_2, x_3)) \\
If_{n+1} &= \forall^{+}_{n+2}(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.\mathbf{lam}\ x_3.App(App(App(Lift(\forall^{-}_{n+1}(If_n)), x_1), x_2), x_3))
\end{aligned}$$

In addition, we need to extend the definition of $comp(\cdot;\cdot)$, $trans_0(\cdot;\cdot)$ $trans_1(\cdot;\cdot)$ and $trans_k(\cdot;\cdot)$ (for $k > 1$) as follows:

$$\begin{aligned}
comp(\underline{xfs}; If(v_1, v_2, v_3)) &= \mathbf{if}(comp(\underline{xfs}; v_1), comp(\underline{xfs}; v_2), comp(\underline{xfs}; v_3)) \\
trans_0(\underline{xfs}; \mathbf{if}(e_1, e_2, e_3)) &= \mathbf{if}(trans_0(\underline{xfs}; e_1), trans_0(\underline{xfs}; e_2), trans_0(\underline{xfs}; e_3)) \\
trans_1(\underline{xfs}; \mathbf{if}(e_1, e_2, e_3)) &= If(trans_1(\underline{xfs}; e_1), trans_1(\underline{xfs}; e_2), trans_1(\underline{xfs}; e_3)) \\
trans_k(\underline{xfs}; \mathbf{if}(e_1, e_2, e_3)) &= (\text{for } k > 1) \\
&\quad \forall^{-}_{k+1}(If)(trans_k(\underline{xfs}; e_1))(trans_k(\underline{xfs}; e_2))(trans_k(\underline{xfs}; e_3))
\end{aligned}$$

The rest is then a routine check to verify that Lemma 3.3 holds, which subsequently implies Theorem 3.4.

It is completely straightforward to extend λ_{code} with pairs and we omit the related details. In order to support references, all we need is to assume the existence of a type constructor $ref(\cdot)$ and the following functions of the given c-types,

$$\begin{aligned}
ref &: \forall\alpha.(\alpha) \Rightarrow ref(\alpha) \\
deref &: \forall\alpha.(ref(\alpha)) \Rightarrow \alpha \\
update &: \forall\alpha.(ref(\alpha), \alpha) \Rightarrow \mathbf{1}
\end{aligned}$$

where $\mathbf{1}$ stands for the unit type. As an example, the expression below:

$$`(\mathbf{lam}\ x.\mathbf{lam}\ y.update(y, deref(x)))$$

is translated into the following expression,

$$Lam(Lam(App(App(update_1, One), App(deref_1, Shi(One)))))$$

where $update_1 = \mathbf{lam}\ x_1.\mathbf{lam}\ x_2.update(x_1, x_2)$ and $deref_1 = \mathbf{lam}\ x.deref(x)$.

In Calcagno *et al.* (2003), it is argued that a program corresponding to the following one would cause the problem of free variable evaluation to occur in MetaML (Taha & Sheard, 2000) as the reference r stores some open code when it is dereferenced.

```
let val r = ref `1
```

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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

This, however, cannot occur in λ^{+}_{code} as the above program is ill-typed: If r is assigned the type $ref(\langle \epsilon, int \rangle)$, then it cannot be used to store open code; if r is assigned a type $ref(\langle G, int \rangle)$ for some non-empty type environment 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 deal with pattern matching in code. One possible approach is to translate general pattern matching into the following fixed form of 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 c-types, respectively:

$$\begin{array}{rcl} Inl & : & \forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle \gamma, \alpha_1 \rangle) \Rightarrow \langle \gamma, \alpha_1 + \alpha_2 \rangle \\ Inr & : & \forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle \gamma, \alpha_2 \rangle) \Rightarrow \langle \gamma, \alpha_1 + \alpha_2 \rangle \\ CaseOf & : & \forall\gamma.\forall\alpha_1.\forall\alpha_2.\forall\alpha_3.(\langle \gamma, \alpha_1 + \alpha_2 \rangle, \langle \alpha_1 :: \gamma, \alpha_3 \rangle, \langle \alpha_2 :: \gamma, \alpha_3 \rangle) \Rightarrow \langle \gamma, \alpha_3 \rangle \end{array}$$

The rest becomes straightforward and we omit the details.

5.1 Typeful code representation for general pattern matching

When typeful code representation is concerned, it is certainly appealing to handle general pattern matching directly without the translation mentioned above. We now present as follows an approach to representing code involving general pattern matching. We first extend the syntax of λ_{code} as follows,

$$\begin{array}{lrcl} \text{types} & \tau & ::= & \ldots \mid pat(G_1, \tau, G_2) \mid cla(G, \tau_1, \tau_2) \\ \text{patterns} & p & ::= & x \mid cc(p_1, \ldots, p_n) \\ \text{expressions} & e & ::= & \ldots \mid \textbf{case } e_0 \textbf{ of } (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \end{array}$$

where $pat(\cdot,\cdot,\cdot)$ and $cla(\cdot,\cdot,\cdot)$ are two new type constructors. Given a pattern p, we use $pvs(p)$ for the sequence of variables $x_1, \ldots, x_n$ occurring in p, from left to right. As usual, we assume that each variable may occur at most once in a pattern. Given $G, \tau, \tau_1, \ldots, \tau_n$, the type $pat(G, \tau, \tau_n :: \ldots :: \tau_1 :: G)$ is for an expression that represents a pattern p such that $pvs(p) = x_1, \ldots, x_n$ and p is assigned the type τ if x_i are assigned types τ_i for $1 \leqslant i \leqslant n$. Given G, τ_1, τ_2, the type $cla(G, \tau_1, \tau_2)$ is for an expression that represents a clause $p \Rightarrow e$ such that the representation for p is assigned the type $pat(G, \tau_1, G')$ for some G' and the representation for e is assigned the type $\langle G', \tau_2 \rangle$.

We now introduce some constructors for constructing expressions that represent patterns. We use *var* for representing a variable pattern:

$$var \quad : \quad \forall\gamma.\forall\alpha.() \Rightarrow pat(\gamma, \alpha, \alpha :: \gamma)$$

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

Given a constructor cc of type $\forall\Delta_0.(\tau_1,\ldots,\tau_n)\Rightarrow\tau$, we introduce a new constructor $\underline{cc}$:

$$\underline{cc} \quad : \quad \forall\Delta_0.\forall\gamma_0\ldots\forall\gamma_n.(pat(\gamma_0,\tau_1,\gamma_1),\ldots,pat(\gamma_{n-1},\tau_n,\gamma_n))\Rightarrow pat(\gamma_0,\tau,\gamma_n)$$

where for $0\leqslant i\leqslant n$, γ_i are assumed to have no occurrences in Δ_0. Let p be a pattern of the form $cc(p_1,\ldots,p_n)$, the $\underline{cc}(e_1,\ldots,e_n)$ is the representation for p, where $e_1,\ldots,e_n$ are assumed to be the representations for $p_1,\ldots,p_n$, respectively.

The following constructor *Clause* is for constructing expressions that represent clauses:

$$Clause \quad : \quad \forall\gamma.\forall\alpha_1.\forall\gamma'.\forall\alpha_2.(pat(\gamma,\alpha_1,\gamma'),\langle\gamma',\alpha_2\rangle)\Rightarrow cla(\gamma,\alpha_1,\alpha_2)$$

Given a clause $p\Rightarrow e$, $Clause(e_1,e_2)$ is the representation for the clause if e_1 and e_2 are representations for p and e, respectively.

Let *list* be the usual list type constructor, that is, $list(\tau)$ is the type for lists in which each element is of the type τ. We introduce the following code constructor *CaseOf* for constructing typeful code representation for case-expressions:

$$CaseOf \quad : \quad \forall\gamma.\forall\alpha_1.\forall\alpha_2.(\langle\gamma,\alpha_1\rangle,list(cla(\gamma,\alpha_1,\alpha_2)))\Rightarrow\langle\gamma,\alpha_2\rangle$$

We then define $Clause_n$ and $CaseOf_n$ for $n\geqslant 1$ as follows:

$$\begin{aligned}
Clause_n &: \forall\gamma_1\ldots\forall\gamma_{n-1}.\forall\gamma_n.\forall\alpha_1.\forall\gamma_n'.\forall\alpha_2.\\
&\quad \langle\gamma_1,\ldots,\gamma_{n-1};pat(\gamma_n,\alpha_1,\gamma_n')\rangle\rightarrow\langle\gamma_1,\ldots,\gamma_{n-1},\gamma_n';\alpha_2\rangle\rightarrow\\
&\quad \langle\gamma_1,\ldots,\gamma_{n-1};cla(\gamma_n,\alpha_1,\alpha_2)\rangle\\
Clause_1 &= \forall_4^+(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.Clause(x_1,x_2))\\
Clause_{n+1} &= \forall_{n+4}^+(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.App(App(Lift(\forall_{n+3}^-(Clause_n)),x_1),x_2))\\
CaseOf_n &: \forall\gamma_1\ldots\forall\gamma_{n-1}.\forall\gamma_n.\forall\alpha_1.\forall\alpha_2.\\
&\quad \langle\gamma_1,\ldots,\gamma_{n-1},\gamma_n;\alpha_1\rangle\rightarrow\langle\gamma_1,\ldots,\gamma_{n-1};list(cla(\gamma_n,\alpha_1,\alpha_2))\rangle\rightarrow\\
&\quad \langle\gamma_1,\ldots,\gamma_{n-1},\gamma_n;\alpha_2\rangle\\
CaseOf_1 &= \forall_3^+(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.CaseOf(x_1,x_2))\\
CaseOf_{n+1} &= \forall_{n+3}^+(\mathbf{lam}\ x_1.App(Lift(\forall_{n+2}^-(CaseOf_n)),x))
\end{aligned}$$

We now extend the definition of $trans_0(\cdot;\cdot)$, $trans_1(\cdot;\cdot)$ and $trans_k(\cdot;\cdot)$ for $k>1$. Let $e=\mathbf{case}\ e_0\ \mathbf{of}\ (p_1\Rightarrow e_1\mid\cdots\mid p_n\Rightarrow e_n)$, and $pvs(p_i)=x_{i,1},\ldots,x_{i,k_i}$ for $1\leqslant i\leqslant n$; we define $trans_0(\underline{xfs};e)$ as

$$\mathbf{case}\ trans_0(\underline{xfs};e_0)\ \mathbf{of}\ (p_1\Rightarrow e_1'\mid\cdots\mid p_n\Rightarrow e_n')$$

where for each $1\leqslant i\leqslant k$, $e_i'=trans_0(\underline{xfs},x_{i,1}@0,\ldots,x_{i,k_i}@0;e_i)$; we define $trans_1(\underline{xfs};e)$ as

$$CaseOf(trans_1(\underline{xfs};e_0),[cl_1,\ldots,cl_n])$$

where $[cl_1,\ldots,cl_n]$ stands for the list consisting of $cl_1,\ldots,cl_n$ and for each $1\leqslant i\leqslant n$, $cl_i=Clause(v_{p_i},trans_1(\underline{xfs},x_{i,1}@1,\ldots,x_{i,k_i}@1;e_i))$ and v_{p_i} is the representation for p_i; for $k>1$, we define $trans_k(\underline{xfs};e)$ as

$$\forall_{k+2}^-(CaseOf_k)(trans_k(\underline{xfs};e_0))([cl_1^k,\ldots,cl_n^k]_{k-1})$$

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

```
fun listInnerProd v1 v2 =
  let
     fun aux v1 v2 sum =
       case v1 of
           nil => sum
         | cons (x1, v1) =>
            (case v2 of
                 nil => sum
               | cons (x2, v2) => aux v1 v2 (x1 * x2 + sum))
     withtype int list -> int list -> int -> int
  in
     aux v1 v2 0
  end
withtype int list -> int list -> int

fun genListInnerProd v1 =
  let
     fun aux v1 v2 sum =
       case v1 of
           nil => sum
         | cons (x1, v1) =>
            `(case ^v2 of
                  nil => ^sum
                | cons (x2, v2) => ^(aux v1 `v2 `(^x1 * x2 + ^sum)))
     withtype {g} int list -> <g; int list> -> <g; int> -> <g; int>
  in
     `(fn v2 => ^(aux v1 `v2 0))
  end
withtype {g} int list -> <g; int list -> int>
```

Fig. 12. A meta-programming example involving pattern matching at level 1.

where $[cl_1^k, \ldots, cl_n^k]_{k-1}$ is a list at level $k-1$ consisting of elements $cl_1^k, \ldots, cl_n^k$, and for each $1 \leqslant i \leqslant n$,

$$cl_i^k = \forall_{k+3}^{-}(Clause_k)(Lift^{k-1}(v_{p_i}))(trans_k(\underline{xfs}, x_{i,1}@k, \ldots, x_{i,k_i}@k; e_i)).$$

It should be clear how the definition of $comp(\cdot,\cdot)$ needs to be extended, and we omit the further details.

Lastly, we present an example in Figure 12, which involves the use of pattern matching at level 1: The function *listInnerProd* computes the inner product of two integer vectors represented as two integer lists, and the function *genListInnerProd* is a staged implementation such that given an integer list v_1, $genListInnerProd(v_1)$ returns the representation for a program that computes $listInnerProd(v_1)(v_2)$ when given an integer list v_2.

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.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

An early reference to partial evaluation can be found in Futamura (1971), 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 Ershov (1977) and later expanded into multi-level staged computation (Jones *et al.*, 1985; Glück & Jørgensen, 1997). 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 Davies & Pfenning (2001), 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
```

Note that *power1*(2) essentially generates the following code:

```
'(fn x => x * (fn x => x * (fn x => 1) x) x)
```

while *power2*(2) outputs the following code:

```
'(fn x => x * (x * 1))
```

So normally *power2* is much more desirable than *power1*. However, the function *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 Davies (1996), 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.[7]

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

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

This approach is essentially used in the development of MetaML (Taha & Sheard, 2000), an extension of ML that supports typed meta-programming by allowing the programmer to manually stage programs with explicit staging annotations. On the 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 Moggi *et al.* (1999), 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 (Calcagno *et al.*, 2003) 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:[8]

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

However, e_2 cannot be assigned a type in Moggi *et al.* (1999) or Calcagno *et al.* (2003), 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 Taha & Nielsen (2003) 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 (Launchbury & Peyton-Jones, 1995). 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}. However, further study indicates that there are some semantic mismatches between λ^{+}_{code} and the type system in Taha & Nielsen (2003), making an encoding of classifiers in λ^{+}_{code} unlikely (unless certain significant modifications are made on classifiers).

[8] 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** $x.x$.

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

Another approach to addressing the limitation of $\lambda^{\Box}$ is presented in Nanevski & Pfenning (n.d.). Instead of refining the notion of (potentially open) code in $\lambda^{\bigcirc}$, the calculus $\nu^{\Box}$ in Nanevski & Pfenning (n.d.) 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 (Pitts, 2003) and FreshML (Pitts & Gabbay, 2000). 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.

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 Pfenning & Lee (1989), 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 Danvy & Rhiger (2001), 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}).

Template Haskell is a recent extension of Haskell (Sheard & Peyton Jones, 2002) with support for *compile-time* meta-programming. Unlike in λ^{+}_{code}, programs generated at compile-time (from a well-typed program) in Template Haskell are not guaranteed to be well-typed in Haskell. Instead, each generated program needs to be type-checked. On the other hand, Template Haskell is rather flexible in allowing the construction of many useful programs that cannot be done in λ^{+}_{code} because of the restriction of the type system of λ^{+}_{code}. For instance, Template Haskell allows a direct implementation of the *printf* function that supports more or less the same syntax in C.

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 indexes. We form a language λ_{code} in which

https://doi.org/10.1017/S0956796805005617 Published online by Cambridge University Press

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 $\langle\langle \% e \rangle\rangle$ (Taha & Nielsen, 2003) can be readily explained in λ_{code}, where $\langle e \rangle$ corresponds to the notation ‘(e) in λ^{+}_{code}; the former and the latter are translated into $Lift(e')$ and $App(Lift(lift), e')$, respectively, where e' is the translation $trans_1(\emptyset; e)$ 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 integrating this approach to meta-programming through typeful code representation into a full-fledged functional programming language, making it available for the purpose of practical programming.

Acknowledgments

We thank Walid Taha for his valuable comments on a previous version of the paper and Joseph Hallett and Rui Shi for their efforts on proofreading the paper.

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

References

Baars, A. I. and Swierstra, S. D. (2002) Typing Dynamic Typing. *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, pp. 157–166.

Baars, A. I. and Swierstra, S. D. (2004) Type-safe, self-inspecting code. *Proceedings of 2004 Haskell Workshop*, pp. 69–79.

Calcagno, C., Moggi, E. and Sheard, T. (2003) Closed Types for a Safe Imperative MetaML. *J. Funct. Program.* (to appear).

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

Chen, C. and Xi, H. (2003) Implementing Typeful Program Transformations. *Proceedings ACM Sigplan Workshop on Partial Evaluation and Semantics based Program Manipulation*, pp. 20–28.

Chen, C., Shi, R. and Xi, H. (2004a) *Distributed meta-programming*. Available at `http://www.cs.bu.edu/~hwxi/academic/drafts/DMP.ps`.

Chen, C., Zhu, D. and Xi, H. (2004b) Implementing cut elimination: a case study of simulating dependent types in Haskell. *Proceedings of 6th International Symposium on Practical Aspects of Declarative Languages: LNCS 3057*. Springer-Verlag.

Church, A. (1940) A formulation of the simple type theory of types. *J. Symbolic Logic*, **5**, 56–68.

Damas, L. and Milner, R. (1982) Principal type-schemes for functional programs. *Proceedings 9th Annual ACM Symposium on Principles of Programming Languages (POPL '82)*, pp. 207–212.

Danvy, O. and Rhiger, M. (2001) A simple take on typed abstract syntax in Haskell-like languages. *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS '01)*, pp. 343–358.

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

Davies, R. and Pfenning, F. (2001) A Modal Analysis of Staged Computation. *J. ACM*, **48**(3), 555–604.

de Bruijn, N. G. (1972) Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, **34**, 381–392.

Deutsch, L. P. and Schiffman, A. M. (1984) Efficient Implementation of Smalltalk-80 System. *Proceedings of 11th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 297–302.

Draves, S. (1998) Partial evaluation for media processing. *ACM Comput. Surv. (CSUR)*, **30**(3es), 21.

Dybvig, R. K. (1992) *Writing hygienic macros in Scheme with syntax-case*. Technical Report #356. Indiana University.

Ershov, A. P. (1977) On the partial computation principle. *Infor. Process. Lett.* **6**(2), 38–41.

Futamara, Y. (1971) Partial evaluation of computation process. *Syst., Comput., Controls*, **2**(5), 45–50.

Glück, R. and Jørgensen, J. (1997) An Automatic Program Generator for Multi-Level Specialization. *Lisp & Symbolic Computation*, **10**(2), 113–158.

Hinze, R. (2001) Manufacturing Datatypes. *J. Funct. Program.* **11**(5), 493–524.

Jones, N. D., Sestoft, P. and Søndergaard, H. (1985) An experiment in partial evaluation: the generation of a compiler generator. *Rewriting Techniques and Applications: LNCS 202*, pp. 124–140. Springer-Verlag.

Launchbury, J. and Peyton-Jones, S. (1995) State in Haskell. *Lisp & Symbolic Computation*, 293–342.

Leone, M. and Lee, P. (1996) Optimizing ML with run-time code generation. *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 137–148. ACM Press.

Massalin, H. (1992) *An Efficient Implementation of Fundamental Operating System Services*. P D dissertation, Columbia University.

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

Moggi, E., Taha, W., Benaissa, Z.-El-Abidine and Sheard, T. (1999) An Idealized MetaML: simpler, and more expressive. *European Symposium on Programming (ESOP '99): LNCS 1576*, pp. 193–207. Springer-Verlag.

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

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

Pfenning, F. and Elliott, C. (1988) Higher-order abstract syntax. *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pp. 199–208.

Pfenning, F. and Lee, P. (1989) A language with Eval and polymorphism. *International Joint Conference on Theory and Practice in Software Development: LNCS 352*, pp. 345–359. Springer-Verlag.

Pike, R., Locanthi, B. and Reiser, J. (1985) Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software – Pract. & Exper* **15**(2), 131–151.

Pitts, A. M. and Gabbay, M. J. (2000) A metalanguage for programming with bound names modulo renaming. In: Backhouse, R. and Oliveira, J. N. (editors), *Proceedings of the 5th International Conference on Mathematics of Program Construction, (MPC '00): LNCS 1837*, pp. 230–255. Springer-Verlag.

Pitts, A. M. (2003) Nominal logic, a first order theory of names and binding. *Infor. & Computation*, **186**(2), 165–193.

Sheard, T. and Peyton Jones, S. (2002) Template metaprogramming for Haskell. In:Chakravarty, M. M. T. (editor), *ACM SIGPLAN Haskell Workshop 02*, pp. 1–16. ACM Press.

Sheard, T., Taha, W., Benaissa, Z. and Pasalic, E. (n.d.) *MetaML.* Available at `http://www.cse.ogi.edu/PacSoft/projects/metaml/`.

Taha, W. and Nielsen, M. F. (2003) Environment classifiers. *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 26–37.

Taha, W. and Sheard, T. (1997) Multi-stage programming with explicit annotations. *Proceedings of the Symposium on Partial Evaluation and Semantic-based Program Manipulation (PEPM '97)*, pp. 203–217.

Taha, W. and Sheard, T. (2000) MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* **248**(1-2), 211–242.

Taha, W., Calcagno, C., Huang, L. and Leroy, X. (n.d.) *MetaOCaml.* Available at `http://www.cs.rice.edu/~taha/MetaOCaml/`.

Wright, A. (1995) Simple imperative polymorphism. *J. Lisp & Symbolic Computation*, **8**(4), 343–355.

Xi, H., Chen, C. and Chen, G. (2003) Guarded recursive datatype constructors. *Proceedings 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 224–235. ACM Press.