First-Class Open and Closed Code Fragments

Morten Rhiger

Department of Computer Science
Roskilde University, Denmark

E-mail: mir@ruc.dk
Homepage: www.ruc.dk/~mir/

Abstract

Staged languages that allow "evaluation under lambdas" are excellent implementation languages for programs that manipulate, specialize, and execute code at runtime. In statically typed staged languages, the existence of staging primitives demands a sound distinction between open code (that may be manipulated under lambdas) and closed code (that may be executed). We present $\lambda^{[]}$, a monomorphic type-safe staged language for manipulating code with free identifiers. It differs from most existing staged languages (such as, for example, derivatives of MetaML) in that its dynamic fragment is not hygienic; in other words, dynamic identifiers are not renamed during substitution. $\lambda^{[]}$ contains a first-class run operation, supports mutable cells (and other computational effects), and has decidable type inference. As such, it is a promising first step towards a practical multi-stage programming language.

1 INTRODUCTION

The field of staged programming grew out of the studies of offline partial evaluation conducted by Jones's group in the late 1980's [3, 13, 16]. Offline partial evaluation is a technique for specializing programs to parts of their inputs [6]. In offline partial evaluation, a program subject to specialization is first binding-time separated by differentiating between its static program parts (those that can be "specialized away" when the static inputs is supplied) and dynamic program parts (those that must remain in the specialized program). The binding-time separated program is then executed on its static inputs by interpreting the dynamic program parts as code generating primitives. The output of such a staged program is (the text of) the specialized program. When viewed operationally like this, a binding-time separated program is called a generating extension [12]. In offline partial evaluation, binding times are statically verifiable properties that play the same role as types in statically typed languages [21].

In the mid-1990's, Davies [9] and Davies and Pfenning [11] established the logical connections between types and binding times. They showed that in the two calculi $\lambda^{\bigcirc}$ and $\lambda^{\Box}$, images of, respectively, linear-time temporal logic and the intuitionistic modal logic S4 under the Curry-Howard isomorphism, type correctness corresponds to binding-time correctness. These calculi extend the lambda calculus with types $\bigcirc\tau$ and $\Box\tau$ of dynamic program parts of type τ. Operationally, the primary difference between the two calculi is that $\lambda^{\bigcirc}$ allows manipulation of

open code fragments while $\lambda^{\Box}$ only allows manipulation of closed code fragments. On the other hand, a first-class eval-like operation is definable in terms of existing primitives in $\lambda^{\Box}$ while $\lambda^{\bigcirc}$ becomes unsound in the presence of such an operation. Neither of these calculi were designed to handle imperative features.

Sheard and Taha accepted the challenge of extending a complete ML-like language with a type of code fragments and primitives for manipulating them [28]. The resulting language, MetaML, has undergone many revisions and its type system has been extended in many directions [4, 20, 26, 27, 28, 29].

1.1 Hygienic languages

Most existing staged languages, such as those mentioned above, are *hygienic* [18]. That is, dynamic (or future-stage) program parts are lexically scoped. Hygiene is required to guarantee confluence for rewriting systems where a term carrying free variables may be substituted under a binder for one of its free variables. Examples include rewriting systems that performs (full) β-normalization (as performed by, e.g, partial evaluation [16], type-directed partial evaluation [7], Barendregt et al's innermost spine reduction [1], and many other program optimizations), macro systems, staged languages, and variants of Scheme where globally defined symbols are not reduced until they are needed (facilitating incrementally defined mutually recursive procedures [17, Section 5.2.1]).

In macro systems and in staged languages, lexically scoped dynamic identifiers are often implemented by consistently renaming them using, e.g., a gensym-like operation [19]. However, in the presence of side effects and of an eval-like operation it has turned out to be somewhat challenging to design languages that account for renaming of dynamic identifiers in their type systems.

1.2 Contributions

We propose to take *non-hygiene* as the primary principle for designing the staged language $\lambda^{[]}$. Since variables are not renamed, the type system may explicitly mention free variables in the type of dynamic code. This enables, we believe, a fairly straightforward treatment of code and stages in the type system as illustrated below.

An immediate and intriguing consequence of our design is that the type system enables a uniform treatment of open and closed program fragments as *first-class data*. The type system of $\lambda^{[]}$ extends the simply-typed lambda calculus with a type $[\gamma]\tau$ of code. Intuitively, an expression has type $[\gamma]\tau$ if it evaluates to a code fragment which has type τ in type environment γ. The language also contains two staging primitives, $\uparrow$ and $\downarrow$, similar to those found in existing staged languages [9, 28] and to quasiquote and unquote in Lisp and Scheme [2].

In $\lambda^{[]}$, source identifiers occur in type environments and hence also in types. As an extreme example, the expression $\uparrow x$ has type $[x:\mathsf{int}]\mathsf{int}$; indeed $\uparrow x$ evaluates to the program fragment x which certainly has type int in a type environment $x:\mathsf{int}$.

The type $[x:\mathsf{int}]\mathsf{int}$ is not the principal type of $\uparrow x$. This expression also has type, say, $[x:\mathsf{bool}, y:\beta]\mathsf{bool}$ and neither of these types appear to be more general than the other. This is mirrored by the fact that neither $x:\mathsf{int} \vdash x:\mathsf{int}$ nor $x:\mathsf{bool}, y:\beta \vdash x:\mathsf{bool}$ are *principal typings* [15] of the expression x. Principal types that involve code types use *type environment variables* γ in a way similar to the use of traditional *type variables*. For example, the principal type of $\uparrow x$ is $[x:\alpha, \gamma]\alpha$; indeed $\uparrow x$ evaluates to the program fragment x which has type α in *any* type environment of shape $x:\alpha, \gamma$. We can replace γ and α by int and $\emptyset$ (where $\emptyset$ denotes the empty type environment) or by $y:\beta$ and bool to obtain the types presented above. Since type environments are present in the type of code, the type of *closed code* of type τ is easily expressible as $[\emptyset]\tau$ which we also write as $[]\tau$. Therefore, a first-class run operation has type $[]\tau \to \tau$.

The type system alleviates some of the weaknesses of non-hygiene. For example, it is sometimes possible to read from the types reported by the type system, the variables that may be captured during evaluation. For example, the type of $\lambda c.\uparrow(\mathsf{let}\, x = 42\, \mathsf{in} \downarrow c)$,

$$[x:\mathsf{int}, \gamma]\tau \to [\gamma]\tau$$

shows that this expression denotes a function that maps code into code, and that any free x's in the input will be captured in the output.

1.3 Outline

We present the terms, type system, and operational semantics of $\lambda^{[]}$ in Section 2. In section 3 we present a proof of type safety. In Section 4 we demonstrate the capabilities of the proposed language using examples. We also take the liberty to introduce mutable cells in this section, although they have not been dealt with in the proof of type safety. In Section 5 we outline related work in the area of staged programming and in Section 6 we conclude.

2 OPEN AND CLOSED CODE FRAGMENTS

In this section we introduce the syntax, the type system, and an operational semantics of the monomorphic staged language $\lambda^{[]}$. We let x, y, z range over an infinite set $\mathbf{V}$ of identifiers. Furthermore we let i range over the set $\mathbf{Z}$ of integers and n over the set $\mathbf{N} = \{0, 1, \cdots\}$ of natural numbers.

The terms of $\lambda^{[]}$ are defined by the following grammar.

$$e, E \quad \in \quad \text{EXPR} \quad ::= \quad i \mid x \mid \lambda x.e \mid e_1\, e_2 \mid \uparrow e \mid \downarrow e \mid \mathsf{lift}\, e \mid \mathsf{run}\, e$$

The operations $\uparrow$, $\downarrow$, and run correspond, respectively, to `quasiquote`, `unquote`, and `eval` in Lisp and Scheme. The expression $\mathsf{lift}\, e$ injects the value of e into a future stage. There is no general lifting operation in Lisp and Scheme; however, the value of any "self-evaluating" constant expression is treated (by `eval`) as the constant expression itself [17].

2.1 Type system

The types of the staged language consist of base types (here restricted to one type int of integers), function types, and a type $[\gamma]\tau$ for code fragments of type τ in type environment γ.

$$\begin{array}{rcrcl} \tau & \in & \textsc{Type} & ::= & \mathsf{int} \mid \tau_1 \to \tau_2 \mid [\gamma]\tau \\ \gamma & \in & \textsc{Env} & ::= & \emptyset \mid x{:}\tau, \gamma \\ & & \Gamma & ::= & \gamma_1, \cdots, \gamma_n \end{array}$$

In the type rules and during type checking, a stack Γ contains type environments mapping free identifiers to types. Free static identifiers are at the bottom of the stack (to the left) while dynamic identifiers are further towards the top (to the right). In contrast to traditional typed lambda calculi, type environments not only exists during type checking but may also appear in the final types assigned to terms. In the presentation of the type system, we represent type environments uniformly as elements of the inductively defined set ENV. In order to achieve principality [14], however, care must be taken to identify type environments that contain re-ordered bindings rather than only syntactically identical type environments. For example, the term $\uparrow(f\,x)$ can be assigned both of the types $[x{:}\tau_1, f{:}\tau_1 \to \tau_2]\tau_2$ and $[f{:}\tau_1 \to \tau_2, x{:}\tau_1]\tau_2$ both of which we treat as principal type of $\uparrow(f\,x)$. Technically, we define an equivalence relation on types and type environment that take re-ordering of bindings into account and we work on the equivalence classes induced by this relation instead of directly on the inductively defined elements.

Definition 1. *We obtain a finite mapping* $\mathbf{V} \to \textsc{Type}$*, here expressed as a subset of* $\mathbf{V} \times \textsc{Type}$*, from a type environment as follows.*

$$\begin{array}{c} \overline{\emptyset} = \{\,\} \\ \overline{x{:}\tau, \gamma} = \{(x,\tau)\} \cup \{(y,\tau') \in \overline{\gamma} \mid y \neq x\} \end{array}$$

This definition guarantees that $\overline{\gamma}$ is a function, i.e., a subset of $\mathbf{V} \times \textsc{Type}$ for which $(x,\tau_1) \in \overline{\gamma}$ and $(x,\tau_2) \in \overline{\gamma}$ implies $\tau_1 = \tau_2$. Note that bindings to the left take precedence over bindings to the right.

We treat two type environments as equivalent if they denote the same finite mapping. This notion of equivalence is extended to types as follow.

Definition 2 (Type equivalence). *We define an equivalence relation* $\approx$ *on types and type environments as follows.*

$$\frac{dom(\overline{\gamma_1}) = dom(\overline{\gamma_2}) \quad \text{For all } x \in dom(\overline{\gamma_1}),\ \overline{\gamma_1}(x) \approx \overline{\gamma_2}(x)}{\gamma_1 \approx \gamma_2} \qquad \frac{}{\mathsf{int} \approx \mathsf{int}}$$

$$\frac{\tau_1 \approx \tau_2 \quad \tau_1' \approx \tau_2'}{\tau_1 \to \tau_1' \approx \tau_2 \to \tau_2'} \qquad \frac{\gamma_1 \approx \gamma_2 \quad \tau_1 \approx \tau_2}{[\gamma_1]\tau_1 \approx [\gamma_2]\tau_2}$$

$$\frac{}{\Gamma,\gamma\vdash i:\mathsf{int}}\ (\text{T-INT}) \qquad \frac{\Gamma,\gamma\vdash e:\tau}{\Gamma\vdash\uparrow e:[\gamma]\tau}\ (\text{T-UP})$$

$$\frac{\overline{\gamma}(x)\approx\tau}{\Gamma,\gamma\vdash x:\tau}\ (\text{T-VAR}) \qquad \frac{\Gamma\vdash e:[\gamma]\tau}{\Gamma,\gamma\vdash\downarrow e:\tau}\ (\text{T-DOWN})$$

$$\frac{\Gamma,\gamma'\vdash e:\tau \quad \gamma'\approx(x{:}\tau',\gamma)}{\Gamma,\gamma\vdash\lambda x.e:\tau'\to\tau}\ (\text{T-ABS}) \qquad \frac{\Gamma\vdash e:\tau}{\Gamma\vdash\mathsf{lift}\,e:[\gamma]\tau}\ (\text{T-LIFT})$$

$$\frac{\Gamma,\gamma\vdash e_1:\tau_2\to\tau \quad \Gamma,\gamma\vdash e_2:\tau_2}{\Gamma,\gamma\vdash e_1\,e_2:\tau}\ (\text{T-APP}) \qquad \frac{\Gamma\vdash e:[]\tau}{\Gamma\vdash\mathsf{run}\,e:\tau}\ (\text{T-RUN})$$

FIGURE 1. Typing rules of the monomorphic, staged language $\lambda^{[]}$.

Note that $\approx$ is indeed an equivalence relation. We extend this notion of equivalence to stacks of type environments: Two stacks of type environments are equivalent, written $\Gamma_1\approx\Gamma_2$, if their elements are point-wise equivalent.

In the rest of this article, we make frequent use of the construction of finite mappings from type environments and of the equality relation. For example, a γ that satisfies the equation

$$[x{:}\tau_1,\gamma]\tau\approx[y{:}\tau_2,\gamma]\tau$$

for $x\neq y$, must have $\{x,y\}\subseteq dom(\overline{\gamma})$ and must already assign τ_1 to x and τ_2 to y. In other words, $\gamma\approx[x{:}\tau_1,y{:}\tau_2,\gamma']$ for some γ'.

We use the following notational conventions. We omit the empty type environment $\emptyset$ inside brackets and write, for example, $[]\tau$ and $[x{:}\tau]\tau$ instead of $[\emptyset]\tau$ and $[x{:}\tau,\emptyset]\tau$. The code-type constructor $[-]-$ has higher precedence than function arrows, so, for example, the types $[]\tau\to\tau$ and $([]\tau)\to\tau$ are identical and both different from $[](\tau\to\tau)$. We write ε for the empty stack of type environments. We allow access to elements at both end of the stack; thus γ,Γ denotes a stack with γ at the bottom while Γ,γ denotes a stack with γ at the top. The operation $|-|$ yields the height of a stack of type environments, i.e., $|\gamma_1,\cdots,\gamma_n|=n$ and $|\varepsilon|=0$.

The type system for the monomorphic languages is given in Figure 1. Modulo type equivalence, the premise of the rule T-ABS may be read simply as the more traditional $\Gamma,(x{:}\tau',\gamma)\vdash e:\tau$. The rule T-RUN states that running a term of code type $[]\tau$ yields a value of type τ. It is intended to guarantee that only closed and well-typed code fragments can be executed at runtime. The symbol run may be abstracted, as in $\lambda x.\mathsf{run}\,x$, but, like any other function, the resulting function of type $[]\tau\to\tau$ cannot be used polymorphically in this monomorphic language. Note that stages participate in relating the definition and the uses of a variable in the sense that identifiers at different stages are always different. For example, the expression $\lambda x.\uparrow x$ contains two distinct identifiers of name x, the rightmost unbound. This ex-

pressions is equivalent to $\lambda y.{\uparrow}x$. Similarly, $\lambda x.{\downarrow}x$ contains two distinct identifiers of name x. (This expression is not equivalent to $\lambda y.{\downarrow}x$, however). Therefore, $(\lambda x.{\downarrow}x)\,E$ does not reduce to ${\downarrow}E$.

To justify the definition of type equivalence, we show that if a term can be given a type then it can also be given any equivalent type.

Lemma 3. *If* $\Gamma_1 \vdash e : \tau_1$, $\Gamma_1 \approx \Gamma_2$, *and* $\tau_1 \approx \tau_2$ *then* $\Gamma_2 \vdash e : \tau_2$.

Proof. By induction on the derivation of $\Gamma_1 \vdash e : \tau_1$.

2.2 Operational semantics

We present a left-to-right, call-by-value, small-step operational semantics of $\lambda^{[]}$ below. The key difference between this semantics and the semantics of hygienic multi-staged languages is that substitution does not rename bound dynamic identifiers. (It does rename bound static identifiers, however.)

We first need the following auxiliary definition.

Definition 4 (Free variables). $\mathrm{FV}(e)_n$ *denotes the set of free static identifiers in the stage-n term e.*

$$
\begin{array}{rclcrcl}
\mathrm{FV}(i)_n & = & \{\,\} & & & & \\
\mathrm{FV}(x)_0 & = & \{x\} & & \mathrm{FV}(x)_{n+1} & = & \{\,\} \\
\mathrm{FV}(\lambda x.e)_0 & = & \mathrm{FV}(e)_0 \backslash \{x\} & & \mathrm{FV}(\lambda x.e)_{n+1} & = & \mathrm{FV}(e)_{n+1} \\
\mathrm{FV}(e_1\,e_2)_n & = & \mathrm{FV}(e_1)_n \cup \mathrm{FV}(e_2)_n & & & & \\
\mathrm{FV}(\uparrow e)_n & = & \mathrm{FV}(e)_{n+1} & & \mathrm{FV}(\downarrow e)_{n+1} & = & \mathrm{FV}(e)_n \\
\mathrm{FV}(\mathsf{lift}\,e)_n & = & \mathrm{FV}(e)_n & & \mathrm{FV}(\mathsf{run}\,e)_n & = & \mathrm{FV}(e)_n
\end{array}
$$

In order to treat the manipulation of identifiers carefully, substitution is defined as a relation on raw terms rather than as a function on α-equivalence classes as follows. As in traditional λ-calculi, fresh static identifiers are introduced to avoid capturing free variable.

Definition 5 (Substitution).

$$
\begin{array}{rcl}
i\{E/x\}_n & = & i \\
x\{E/x\}_0 & = & E \\
y\{E/x\}_0 & = & z, \quad \textit{for } x \neq y \\
(\lambda x.e)\{E/x\}_0 & = & \lambda x.e \\
(\lambda z.e)\{E/x\}_{n+1} & = & \lambda z.e\{E/x\}_{n+1} \\
(\lambda z.e)\{E/x\}_0 & = & \lambda y.e\{y/z\}_0\{E/x\}_0, \\
 & & \textit{for } z \neq x \textit{ and } y \notin (\mathrm{FV}(e)_0\{z\}) \cup \mathrm{FV}(E)_0 \\
(e_1\,e_2)\{E/x\}_n & = & (e_1\{E/x\}_n)\,(e_2\{E/x\}_n) \\
(\uparrow e)\{E/x\}_n & = & \uparrow e\{E/x\}_{n+1} \\
(\downarrow e)\{E/x\}_{n+1} & = & \downarrow e\{E/x\}_n \\
(\mathsf{lift}\,e)\{E/x\}_n & = & \mathsf{lift}\,e\{E/x\}_n \\
(\mathsf{run}\,e)\{E/x\}_n & = & \mathsf{run}\,e\{E/x\}_n
\end{array}
$$

$$\frac{}{V \vdash_n i}\ (\text{V-INT}) \qquad \frac{V \vdash_{n+1} e}{V \vdash_n \uparrow e}\ (\text{V-UP})$$

$$\frac{}{V \vdash_{n+1} x}\ (\text{V-VAR}) \qquad \frac{V \vdash_{n+1} e}{V \vdash_{n+2} \downarrow e}\ (\text{V-DOWN})$$

$$\frac{}{V \vdash_0 \lambda x.e}\ (\text{V-ABS-1}) \qquad \frac{V \vdash_{n+1} e}{V \vdash_{n+1} \mathsf{lift}\, e}\ (\text{V-LIFT})$$

$$\frac{V \vdash_{n+1} e}{V \vdash_{n+1} \lambda x.e}\ (\text{V-ABS-2}) \qquad \frac{V \vdash_{n+1} e}{V \vdash_{n+1} \mathsf{run}\, e}\ (\text{V-RUN})$$

$$\frac{V \vdash_{n+1} e_1 \quad V \vdash_{n+1} e_2}{V \vdash_{n+1} e_1\, e_2}\ (\text{V-APP})$$

FIGURE 2. Values of $\lambda^{[]}$.

Restricted to operate on static terms (e.g., $-\{-/-\}_0$), this notion of substitution coincides with that traditionally defined for the λ-calculus. For the dynamic fragment, identifiers are dynamically bound in that they are not renamed during substitution. For example, substituting $\uparrow y$ for the free static identifier x in $\uparrow(\lambda y.\downarrow x)$ does not rename the bound dynamic identifier y: $(\uparrow(\lambda y.\downarrow x))\{\uparrow y/x\}_0 = \uparrow(\lambda y.\downarrow\uparrow y)$.

The following two fundamental properties of the simply-typed λ-calculus also hold for $\lambda^{[]}$.

Lemma 6 (Weakening). *If* $\gamma, \Gamma \vdash e : \tau$ *and* $x \notin \mathrm{FV}(e)_{|\Gamma|}$ *then* $(x{:}\tau', \gamma), \Gamma \vdash e : \tau$.

Proof. By induction on the derivation of $\gamma, \Gamma \vdash e : \tau$.

Lemma 7 (Substitution). *If* $(x{:}\tau', \gamma), \Gamma \vdash e : \tau$ *and* $\gamma, \Gamma \vdash e' : \tau'$ *then*

$$\gamma, \Gamma \vdash e\{e'/x\}_{|\Gamma|} : \tau.$$

Proof. By induction on the derivation of $(x{:}\tau', \gamma), \Gamma \vdash e_1 : \tau$. Lemma 6 (Weakening) is used in the case $(x{:}\tau', \gamma) \vdash \lambda z.e : \tau'' \to \tau$ for $z \neq x$.

Figure 2 characterizes the values that can result from evaluating a stage-*n* term. In the following section we show that only these values can be the final results of evaluation well-typed $\lambda^{[]}$-terms. The left-to-right, call-by-value, small-step operational semantics is presented in Figure 3. The rule S-LAM represents evaluation under (dynamic) lambdas.

Contextual rules:

$$\frac{S \vdash_{n+1} e \longrightarrow e'}{S \vdash_{n+1} \lambda x.e \longrightarrow \lambda x.e'} \text{(S-LAM)} \qquad \frac{S \vdash_{n+1} e \longrightarrow e'}{S \vdash_{n} \uparrow e \longrightarrow \uparrow e'} \text{(S-UP)}$$

$$\frac{S \vdash_{n} e_1 \longrightarrow e_1'}{S \vdash_{n} e_1\, e_2 \longrightarrow e_1'\, e_2} \text{(S-APP-1)} \qquad \frac{S \vdash_{n} e \longrightarrow e'}{S \vdash_{n+1} \downarrow e \longrightarrow \downarrow e'} \text{(S-DOWN)}$$

$$\frac{V \vdash_{n} e_1 \quad S \vdash_{n} e_2 \longrightarrow e_2'}{S \vdash_{n} e_1\, e_2 \longrightarrow e_1\, e_2'} \text{(S-APP-2)} \qquad \frac{S \vdash_{n} e \longrightarrow e'}{S \vdash_{n} \mathsf{lift}\, e \longrightarrow \mathsf{lift}\, e'} \text{(S-LIFT)}$$

$$\frac{S \vdash_{n} e \longrightarrow e'}{S \vdash_{n} \mathsf{run}\, e \longrightarrow \mathsf{run}\, e'} \text{(S-RUN)}$$

Reduction rules:

$$\frac{V \vdash_{0} e_2 \quad e_1\{e_2/x\}_0 \gtrsim e_1'}{S \vdash_{0} (\lambda x.e_1)\, e_2 \longrightarrow e_1'} \text{(S-LAM-R)} \qquad \frac{V \vdash_{0} e}{S \vdash_{0} \mathsf{lift}\, e \longrightarrow \uparrow e} \text{(S-LIFT-R)}$$

$$\frac{V \vdash_{1} e}{S \vdash_{1} \downarrow(\uparrow e) \longrightarrow e} \text{(S-DOWN-R)} \qquad \frac{V \vdash_{1} e}{S \vdash_{0} \mathsf{run}\,(\uparrow e) \longrightarrow e} \text{(S-RUN-R)}$$

FIGURE 3. Left-to-right, call-by-value, small-step operational semantics.

3 SYNTACTIC TYPE SOUNDNESS

The following auxiliary result states that a well-typed value at stage $n+1$ is also a well-typed expression at stage n. In that respect, it serves the same purpose as the demotion operation defined for MetaML [20, 27].

Lemma 8 (Demotion). *If* $\emptyset, \Gamma, \gamma \vdash e : \tau$ *and* $V \vdash_{|\Gamma|+1} e$ *then* $\Gamma, \gamma \vdash e : \tau$.

Proof. By induction on the derivation of $\emptyset, \Gamma, \gamma \vdash e : \tau$.

We also need the following result stating that a well-typed value at stage 0 is also a well-typed value at stage 1.

Lemma 9 (Promotion at stage 0). *If* $\emptyset \vdash e : \tau$ *and* $V \vdash_0 e$ *then* $\emptyset, \gamma \vdash e : \tau$.

Proof. Since e is a value at stage 0, e is closed. Hence, by repeating Lemma 6 (Weakening), $\gamma \vdash e : \tau$. Then we also have (by straightforward separate lemma) that $\emptyset, \gamma \vdash e : \tau$ as required.

Lemma 10 (Subject reduction). *If* $\emptyset, \Gamma \vdash e : \tau$ *and* $S \vdash_{|\Gamma|} e \longrightarrow e'$ *then* $\emptyset, \Gamma \vdash e' : \tau$.

Proof. By induction on the derivation of $S \vdash_{|\Gamma|} e \longrightarrow e'$. Lemma 7 (Substitution) is used in the case $S \vdash_0 (\lambda x.e_1)\, e_2 \longrightarrow e_1'$, Lemma 8 (Demotion) is used in the case $S \vdash_0 \mathsf{run}\,(\uparrow e) \longrightarrow e$, and Lemma 9 (Promotion) is used in the case $S \vdash_0 \mathsf{lift}\, e \longrightarrow \uparrow e$.

Lemma 11 (Well-typed terms are not stuck). *If* $\emptyset, \Gamma \vdash e : \tau$ *then either*

1. $V \vdash_{|\Gamma|} e$, *or*
2. *there exists* e' *such that* $S \vdash_{|\Gamma|} e \longrightarrow e'$.

Proof. By induction on the derivation of $\emptyset, \Gamma \vdash e : \tau$. A simple type inversion result stating that abstractions are the only values of type $\tau_1 \to \tau_2$ and that quoted terms are the only values of type $[\gamma]\tau$ is used in the three cases corresponding to the reduction rules S-LAM-R, S-DOWN-R, and S-RUN-R.

Definition 12 (Evaluation). *We define an iterated reduction relation inductively as follows.*

$$\frac{\text{There are no } e' \text{ such that } e \longrightarrow e'}{e \longrightarrow^* e} \qquad \frac{e \longrightarrow e'' \quad e'' \longrightarrow^* e'}{e \longrightarrow^* e'}$$

We define a partial function EXPR $\to$ EXPR *as follows.*

$$\mathrm{eval}(e) = \begin{cases} e', & \text{if } e \longrightarrow^* e' \text{ and } V \vdash_0 e' \\ \text{undefined}, & \text{otherwise} \end{cases}$$

The following main result states that a well-typed term either diverges or it yields a value of the same type.

Theorem 13 (Strong type safety). *If* $\emptyset \vdash e : \tau$ *and* $\mathrm{eval}(e) = e'$ *then* $V \vdash_0 e'$ *and* $\emptyset \vdash e' : \tau$.

4 EXAMPLES

Generating extensions can be implemented in $\lambda^{[]}$ using the staging primitives $\uparrow$ and $\downarrow$. The specialized programs output by generating extensions can be evaluated using run to produce first-class functions. As an example, consider the following implementation of the linear integer exponentiation function.[1] (The logarithmic exponentiation function is subject to the same considerations [8] but is left out for pedagogical purposes.)

$$\textbf{fun}\ \exp(n,x)\ = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ x \times (\exp(n-1,x))$$

[1] In the rest of this paper, we use an ML-like notation. We also assume the existence of recursion, conditionals, monomorphic let-expressions, tuples, etc. These extensions are straightforward to add to $\lambda^{[]}$.

A generating extension of this function is a program that, given an integer n, yields (the text of) a program that computes x^n. A standard binding-time analysis of the exponentiation function with respect to n being static and x dynamic reveals that the integer constant 1 and the multiplication are dynamic. The rest of the program is static. These annotations give rise to following the staged exponentiation function.

$$\textbf{fun}\ \mathrm{exp}_{\mathrm{bta}}(n,x)\ = \textbf{if}\ n = 0\ \textbf{then}\ \uparrow 1\ \textbf{else}\ \uparrow(\downarrow x \times \downarrow(\mathrm{exp}_{\mathrm{bta}}(n-1,x)))$$

This function has principal type $\mathsf{int} \times [\gamma]\mathsf{int} \to [\gamma]\mathsf{int}$. It is used at type $\mathsf{int} \times [z:\mathsf{int}]\mathsf{int} \to [z:\mathsf{int}]\mathsf{int}$ below. The generating extension is defined as the following two-level eta-expansion of the staged exponentiation function.

$$\textbf{fun}\ \mathrm{exp}_{\mathrm{gen}}(n)\ =\ \uparrow(\lambda z.\downarrow(\mathrm{exp}_{\mathrm{bta}}(n,\uparrow z)))$$

The generating extension $\mathrm{exp}_{\mathrm{gen}}$ has type $\mathsf{int} \to [\,](\mathsf{int} \to \mathsf{int})$. This type shows that $\mathrm{exp}_{\mathrm{gen}}$ produces closed code fragments. For example, $\mathrm{exp}_{\mathrm{gen}}(3)$ has type $[\,](\mathsf{int} \to \mathsf{int})$ and evaluates to a program fragment $\lambda z.z \times z \times z \times 1$. This code fragment can be executed using the operator run. Hence, $\mathsf{run}\,(\mathrm{exp}_{\mathrm{gen}}(3))$ has type $\mathsf{int} \to \mathsf{int}$ and yields a first-class cube operation.

The example above is a standard application of staged languages but it demonstrates an important practical property of staged programs: By erasing the staging primitives one obtains the original unstaged program. In order to apply this principle to programs using mutable cells, the staged program must be able to store open code fragments in mutable cells. Although we have left out the treatment of mutable cells (and other computational effects) of this extended abstract, let us suppose we add the following (monomorphic) operations.

$$\begin{array}{rcl} \textbf{ref} & : & \alpha \to \alpha\,\mathsf{ref} \\ := & : & \alpha\,\mathsf{ref} \times \alpha \to \mathsf{unit} \\ ! & : & \alpha\,\mathsf{ref} \to \alpha \end{array}$$

We can then consider the following imperative exponentiation function of type $\mathsf{int} \times \mathsf{int} \to \mathsf{int}$.

```
fun impexp(n,x) =
    let val m = ref n
        val r = ref 1
    in while !m > 0 do (r := x * !r; m := !m - 1);
        !r
    end
```

A staged version of this functions looks as follows.

```
fun impexp_bta(n,x) =
    let val m = ref n
        val r = ref ↑1
```

```
in  while !m > 0 do (r := ↑(↓x × ↓!r); m := !m − 1);
    !r
end
```

This staged function has principal type $\mathsf{int} \times [\gamma]\mathsf{int} \to [\gamma]\mathsf{int}$. It is, as in the functional case above, used at type $\mathsf{int} \times [z:\mathsf{int}]\mathsf{int} \to [z:\mathsf{int}]\mathsf{int}$ below. The generating extension is defined as above, using a two-level eta-expansion of the binding-time annotated function.

$$\textbf{fun}\ \mathrm{impexp}_{\mathrm{gen}}(n) \ = \ \uparrow(\lambda z.\downarrow(\mathrm{impexp}_{\mathrm{bta}}(n, \uparrow z)))$$

In the function $\mathrm{impexp}_{\mathrm{bta}}$, the variable r has type $([z:\mathsf{int}]\mathsf{int})\,\mathsf{ref}$. During specialization, this cell contains the sequence of dynamic terms 1, $z \times 1$, $z \times z \times 1$, $z \times z \times z \times 1$, etc.

The two examples above motivate the introduction of a general two-level eta expansion

$$\textbf{fun}\ \mathrm{eta}(f) \ = \ \uparrow(\lambda z.\downarrow(f\ \uparrow z))$$

of principal type $([z:\alpha, \gamma]\alpha \to [z:\beta, \gamma]\delta) \to [\gamma](\beta \to \delta)$. The expression

$$\mathrm{eta}(\lambda x.\, \mathrm{exp}_{\mathrm{bta}}(3, x))$$

of type $[\,](\mathsf{int} \to \mathsf{int})$ yields the (text of the) cube function. Note that eta can also be used "non-hygienically." Given $f = \lambda c.\uparrow(\lambda z.\downarrow c)$ of principal type $[z:\alpha, \gamma']\beta \to [\gamma'](\alpha \to \beta)$, $\mathrm{eta}(f)$ evaluates to a representation of the code fragment $\lambda z.\lambda z.z$ in which the inner lambda (supplied by f) shadows the outer lambda (supplied by eta). It is (correctly) assigned the principal type $[\gamma'](\alpha \to \beta \to \beta)$.

5 RELATED WORK

Pfenning and Elliott have used higher-order abstract syntax as a representation of programs (and other entities) with bound variables [23]. Higher-order abstract syntax also enables an embedded lightweight type system for programs that generate code fragments [24]. In the presence of mutable cells (or other sufficiently expressive computational effects), however, higher-order abstract syntax fails to obey standard scoping rules.

The observation that a naive hygienic staged programming language becomes unsound in the presence of a first-class eval operation is due to Rowan Davies, in the context of MetaML. Safe type systems have been designed that combine types for open code and types for closed code and that contain an eval-like operation [4, 20]. We have avoided the complexities of these languages by taking non-hygiene as a primary design principle of $\lambda^{[]}$.

Chen and Xi take a first-order unstaged abstract syntax as the starting point for defining the multi-staged type-safe calculus $\lambda^{+}_{\mathrm{code}}$ [5]. Like $\lambda^{[]}$, $\lambda^{+}_{\mathrm{code}}$ contains a type of code parameterized by a type environment. $\lambda^{+}_{\mathrm{code}}$ is defined by

a translation into λ_{code}, a nameless unstaged extension of the second-order polymorphic λ-calculus. In contrast, $\lambda^{[]}$ is a direct extension of the simply-typed λ-calculus, it does not require polymorphism, and it does not assume de Bruijn-indexed terms. More seriously, dynamic $\lambda^{+}_{\text{code}}$-terms must be closed. For example, the term $\text{let}\, c = \uparrow(x,y)\, \text{in}\, (\lambda x.\lambda y.\downarrow c, \lambda y.\lambda x.\downarrow c)$ is ill-typed in $\lambda^{+}_{\text{code}}$ (because the dynamic sub-term $\uparrow(x,y)$ contains free identifiers) but has type

$$[\gamma](\tau_1 \to \tau_2 \to \tau_1 \times \tau_2) \times [\gamma](\tau_2 \to \tau_1 \to \tau_1 \times \tau_2)$$

in $\lambda^{[]}$.

DynJava, an extension of Java with dynamic compilation, is another type-safe staged language that builds on the principles of non-hygiene [22]. DynJava has only two stages and its code type eliminator (@, which corresponds to the $\downarrow$ of $\lambda^{[]}$) only applies to variables. Furthermore, DynJava is an extension of an imperative, explicitly typed language whereas $\lambda^{[]}$ is an extension of an implicitly typed, higher-order language.

6 CONCLUSIONS

Well-designed languages balance concise specifications against expressive features and safe execution. We have demonstrated that by liberating a staged language from the demand of *hygiene*, it can be given a type system that offers a first-class run operation while requiring only a minimum of extra type constructors and type rules. We also claim, although we have not proved it in this article, that the type system is decidable and that is safely handles mutable cells (and other effects). We have thus provided a first step towards a practical higher-order multi-stage programming language whose type system supports the implementation of run-time specialization and execution and other partial evaluation techniques.

REFERENCES

[1] Henk P. Barendregt, Richard Kennaway, Jan Willem Klop, and M. Ronan Sleep. Needed reduction and spine strategies for the lambda calculus. *Journal of Functional Programming*, 75(3):191–231, 1987.

[2] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, number NS-99-1 in BRICS Note Series, pages 4–12, San Antonio, Texas, January 1999.

[3] Anders Bondorf, Neil D. Jones, Torben Mogensen, and Peter Sestoft. Binding time analysis and the taming of self-application. DIKU Rapport, University of Copenhagen, Copenhagen, Denmark, 1988.

[4] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36, Geneva, Switzerland, July 2000. Springer.

[5] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. *Journal of Functional Programming*, 15(6), 2005.

[6] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[7] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[8] Olivier Danvy. Programming techniques for partial evaluation. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, NATO Science series, pages 287–318. IOS Press Ohmsha, 2000.

[9] Rowan Davies. A temporal-logic approach to binding-time analysis. In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[10] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Steele [25], pages 258–283.

[11] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001. Revised and extended version of [10].

[12] Andrei P. Ershov. On the essence of compilation. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[13] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[14] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[15] Trevor Jim. What are principal typings and what are they good for? In Steele [25], pages 42–53.

[16] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[17] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[18] Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, Indiana, 1986.

[19] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, Cambridge, Massachusetts, August 1986. ACM Press.

[20] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 193–207, Amsterdam, The Netherlands, March 1999.

[21] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[22] Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. DynJava: Type safe dynamic code generation in java. In *The 3rd JSSST Workshop on Programming and Programming Languages*, Tokyo, Japan, March 2001.

[23] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[24] Morten Rhiger. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems*, 25(3):291–315, May 2003.

[25] Guy L. Steele, editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM Press.

[26] Walid Taha. *Multi-Stage programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[27] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, number 1443 in Lecture Notes in Computer Science, pages 918–929. Springer-Verlag, 1998.

[28] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997. ACM Press.

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