Staged Computation with Staged Lexical Scope

Morten Rhiger

Roskilde University, P.O. Box 260, DK-4000 Roskilde, Denmark
mir@ruc.dk

Abstract. We present a simple core type system, $\lambda^{[]}$— pronounced "lambda open box" — for a statically typed, hygienic, and multi-stage lambda-calculus supporting evaluation under future-stage binders, open-code manipulation, a first-class eval function, and mutable state. The type system provides one type of lexically scoped code that precisely accounts for the contexts in which code values can be inserted. In particular, this type can distinguish between open and closed code. We show how to extend $\lambda^{[]}$ with subtype polymorphism over program contexts. The soundness and simplicity of $\lambda^{[]}$ demonstrate that the notion of staging is orthogonal to features that have been presented as instrumental in existing type systems for staged computation, such as polymorphism, nameless term representations, explicit substitutions, and delimited continuations.

1 Introduction

Staged computation enables programs to generate, combine, and execute code values at runtime. Its ability to delay the execution of code values induces a distinction between present-stage (or static) program parts (that are evaluated normally) and future-stage (or dynamic) program parts (that yield code values). Its ability to evaluate under future-stage binders makes staged computation ideal for partial evaluation and program specialization, compilation, runtime code generation, and macro expansion.

Code manipulation as a programming discipline dates back to the development of the first Fortran compiler in the late 1950s [1]. In the early 1960s, McCarthy proposed S-expressions as a uniform representation of code (and other data) in Lisp [15]. The work by the artificial-intelligence community in the 1970s then established *quasi quotations* as the preferred syntactic constructs for building such S-expressions [3]. The development of offline partial evaluation in the 1980s demonstrated that quasi quotations (or similar binding-time annotations) elegantly captures the notion of staged computation [5, 12]. In a couple of influential papers published in the mid 1990s, Davies [9] and Davies and Pfenning [10] established the type-theoretical foundation for staged computation via connections to temporal and modal logics. In the decade that has followed Davies and Pfenning's work, much research have been aimed at designing static type systems that combine general-purpose features with support for staged computation using quasi quotations as code-generation constructs [6, 14, 16, 22, 24, 26].

H. Seidl (Ed.): ESOP 2012, LNCS 7211, pp. 559–578, 2012.

1.1 The Challenge

Statically typed languages that support staged computation must guarantee that well-typed programs only generate, combine, and execute code values that are themselves well typed. Languages that evaluate under future-stage binders demand a careful treatment in the type system of potentially *open code* (that is, code that contains free variables). This is particularly challenging in the presence of an eval function and of assignments to mutable state since (1) an eval function must be passed closed code values only and (2) assignments enable code values to escape the scope in which they are generated, which in turn enables future-stage variables to escape their binder. Hence type systems for staged computation must distinguish between open and closed code values and must prevent future-stage variables from being captured by any binder but their own.

1.2 Our Contributions

We present $\lambda^{[]}$, a sound monomorphic type system for hygienic staged evaluation that supports *multiple stages*, *evaluation under future-stage binders*, and *open-code manipulation*. The type systems provides *one type of code*, which precisely distinguishes open from closed code. The type system supports a *first-class eval function* and *first-class mutable cells*. Mutable cells can contain (open as well as closed) code and assignments can pass (open as well as closed) code across binders. Yet, the type systems prevent future-stage variables from escaping their scope by the means of assignments. We then extend $\lambda^{[]}$ with subtype polymorphism and show that the result, $\lambda^{[]}_{<:}$, is at least as expressive as the foundational multi-stage calculi $\lambda^{\bigcirc}$ [9] and $\lambda^{\Box}$ [10].

The type system of $\lambda^{[]}$ demonstrate that the notion of staging is orthogonal to features that are instrumental in existing type systems for staged computation, such as separate types for closed values [4, 16], polymorphism [25], nameless term representations [8], explicit substitutions [19], and delimited continuations [13]. $\lambda^{[]}$ is the first type system for multi-stage programming that makes an explicit eval function and mutable state coexist with hygienic evaluation under future binders.

2 The Staged Type System $\boldsymbol{\lambda^{[]}}$

$\lambda^{[]}$ is a type system for monomorphic staged λ-calculi. It extends the simply-typed λ-calculus with staging primitives $\uparrow e$ and $\downarrow e$ (similar to **next** and **prev** of $\lambda^{\bigcirc}$ [9] and to brackets $\langle\cdot\rangle$ and escape ~ of the MetaML family of type systems, and reminiscent of `quasiquote` and `unquote` of Lisp and Scheme [3]) and with a single type of code $[\gamma]t$ parametrized over a type environment γ and a type t.

In $\lambda^{[]}$, values of type $[\gamma]t$ are code values: Intuitively, if an expression has type $[\gamma]t$ (and terminates), then it evaluates to (a representation of) a code fragment that has type t under type environment γ. Thus, the code type $[\gamma]t$ precisely characterizes the contexts that code values can be inserted into. Indeed,

by varying γ, this code type is able to characterize both closed and (potentially) open code values. The code types of $\lambda^{[]}$ are *contextual modal types* but the typing rules are different from those of recently developed contextual modal type systems [19]. In terms of temporal logics, $\lambda^{[]}$ models linear (rather than branching) time.

To support hygienic evaluation under future-stage binders, the typing judgment of $\lambda^{[]}$ represents the context of a term by a linearly ordered sequence of type environments,

$$\gamma_0 \cdot \gamma_1 \cdot \ldots \cdot \gamma_{n-1} \cdot \gamma_n \,;\, \gamma_{n+1} \cdot \ldots \cdot \gamma_{m-1} \cdot \gamma_m,$$

of which the type environment to the left of the (unique) ";" is designated as the *current*. The *stage* (or *time*, in the vocabulary of temporal logic) of a bound variable equals the index of the environment that binds it. The stage of an expression is the index of the current type environment. (Lower stages are "more static" or "past"; higher stages are "more dynamic" or "future".) An expression can only access variables of the same stage. Hence variables at different stages live in different *namespaces*. We let Γ range over sequences of type environment not containing ";". We use a single "$\cdot$" to separate elements in a sequence.

When introducing a code value by $\uparrow e$ at stage n, e is typed at stage $n+1$: If $\uparrow e$ is typed in context $\Gamma \,;\, \gamma \cdot \Gamma'$, then e is typed in context $\Gamma \cdot \gamma \,;\, \Gamma'$. By the intuition above, if the type of e is t, then $\uparrow e$ has type $[\gamma]t$. Dually, when eliminating a code value by $\downarrow e$ at stage $n+1$, e is typed at stage n: If $\downarrow e$ is typed in context $\Gamma \cdot \gamma \,;\, \Gamma'$, then e is typed in context $\Gamma \,;\, \gamma \cdot \Gamma'$. Following intuition again, if the type of e is $[\gamma]t$, then $\downarrow e$ has type t. The typing rules for code introduction and elimination concisely sum up this explanation as follows.

$$\frac{\Gamma \cdot \gamma \,;\; \Gamma' \vdash e : t}{\Gamma \;;\gamma \cdot \Gamma' \vdash \uparrow e : [\gamma]t} \qquad ([]\text{-I})$$

$$\frac{\Gamma \;;\gamma \cdot \Gamma' \vdash e : [\gamma]t}{\Gamma \cdot \gamma \,;\; \Gamma' \vdash \downarrow e : t} \qquad ([]\text{-E})$$

The complete type system of $\lambda^{[]}$ is displayed in Fig. 1. The typing rules for variables, abstractions, and application closely mimic the simply-typed λ calculus. They access only the current type environment and pass the remaining sequence of past- and future-stage type environments unmodified to their subterms. The type rules for the staging primitives insist that each $\downarrow$ appears under at least one $\uparrow$. Without this requirement we would be forced to let a static occurrence of $\downarrow$ act as an eval function, but we prefer to study staging using $\uparrow$ and $\downarrow$ on one hand and an eval function on the other in isolation. We write ϵ for the empty sequence of type environments and we let b and c range over an unspecified set of base types and over type-indexed constants, respectively.

The typing rules of $\lambda^{[]}$ define a notion of *staged lexical scope:* A variable is bound by the nearest enclosing definition *at the same stage* of that variable. This notion extends to term variable that appears in type environments in types.

Syntax:

$$\begin{aligned}
\text{(Types)}\quad & t ::= b \mid t \to t \mid [\gamma]t \\
\text{(Terms)}\quad & e ::= c_{\{t\}} \mid x \mid \lambda x\!:\!t.\, e \mid e\, e \mid \uparrow e \mid \downarrow e \\
\text{(Environments)}\quad & \gamma ::= \emptyset \mid \gamma, \gamma \mid x : t \\
\text{(Sequences)}\quad & \Gamma ::= \gamma \cdot \ldots \cdot \gamma
\end{aligned}$$

Typing rules: $\boxed{\Gamma\ ;\ \Gamma' \vdash e : t}$

$$\frac{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash t :: *}{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash c_{\{t\}} : t}\ (\textsc{Const}) \qquad \frac{}{\Gamma \cdot (x : t, \gamma)\ ;\ \Gamma' \vdash x : t}\ (\textsc{Var})$$

$$\frac{\Gamma \cdot \gamma' \cdot \gamma\,;\quad \Gamma' \vdash\ \ e :\quad t}{\Gamma \cdot \gamma'\quad\ ;\gamma \cdot \Gamma' \vdash \uparrow e : [\gamma]t}\ ([]\text{I}) \qquad \frac{\Gamma \cdot (x : t, \gamma)\ ;\ \Gamma' \vdash e : t' \quad \Gamma \cdot \gamma\ ;\ \Gamma' \vdash t :: *}{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash \lambda x\!:\!t.\, e : t \to t'}\ (\to\text{I})$$

$$\frac{\Gamma \cdot \gamma'\quad\ ;\gamma \cdot \Gamma' \vdash\ \ e : [\gamma]t}{\Gamma \cdot \gamma' \cdot \gamma\,;\quad \Gamma' \vdash \downarrow e :\quad t}\ ([]\text{E}) \qquad \frac{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash e_1 : t_2 \to t \quad \Gamma \cdot \gamma\ ;\ \Gamma' \vdash e_2 : t_2}{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash e_1\, e_2 : t}\ (\to\text{E})$$

Kinding rules: $\boxed{\Gamma\ ;\ \Gamma' \vdash t :: \kappa}$ $\boxed{\Gamma\ ;\ \Gamma' \vdash \gamma}$

$$\frac{}{\Gamma\ ;\ \Gamma' \vdash b :: *}\ (\text{K}b) \qquad \frac{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash \gamma' \quad \Gamma \cdot \gamma\ ;\ \Gamma' \vdash t\ :: *}{\Gamma\ ;\ \gamma \cdot \Gamma' \vdash [\gamma']t :: *}\ (\text{K}[])$$

$$\frac{\left.\begin{array}{l}(x : t) \in \gamma, \quad \text{and} \\ \Gamma \cdot \gamma\ ;\ \Gamma' \vdash t :: *\end{array}\right\} \text{ for } (x : t) \in \gamma'}{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash \gamma'}\ (\text{K}\gamma) \qquad \frac{\Gamma\ ;\ \Gamma' \vdash t_0 :: * \quad \Gamma\ ;\ \Gamma' \vdash t_1 :: *}{\Gamma\ ;\ \Gamma' \vdash t_0 \to t_1 :: *}\ (\text{K}\to)$$

Fig. 1. The Type System of $\lambda^{[]}$

Consequently, types that refer to unbound variables or that assert incorrect types for its variables are invalid. The type system of $\lambda^{[]}$ characterizes valid types using the kinding rules of Fig. 1. Most importantly, $[\gamma]t$ is valid at stage n when both γ and t are kind checked at stage $n+1$ since both the type t and any variable bound by γ are future stage entities. For example, $\lambda x\!:\!t.\, \lambda c\!:\![x : t]t.\, c$ is not well typed, since the x inside the asserted type of c is unbound. This x occurs at a stage different from the x bound at the surrounding λ.

2.1 Staged Lexical Scope

Before we present the semantics of $\lambda^{[]}$, we need to extend the definitions of free variables and substitution to a staged setting. This is particularly pertinent in the approach to staged computation we propose, where variables at different stages live in different namespaces.

Definition 1 (Free variables). *The set of free stage-n variables in stage-m terms, types, or environments are defined as follows.*

$$
\begin{aligned}
\mathrm{FV}_n^m(c_{\{t\}}) &= \mathrm{FV}_n^m(t) \\
\mathrm{FV}_m^m(x) &= \{x\} \\
\mathrm{FV}_n^m(x) &= \{\,\}, \quad \textit{if } m \neq n \\
\mathrm{FV}_m^m(\lambda x\!:\!t.\, e) &= \mathrm{FV}_m^m(t) \cup \big(\mathrm{FV}_m^m(e) - \{x\}\big) \\
\mathrm{FV}_n^m(\lambda x\!:\!t.\, e) &= \mathrm{FV}_n^m(t) \cup \mathrm{FV}_n^m(e), \quad \textit{if } m \neq n \\
\mathrm{FV}_n^m(e_1\, e_2) &= \mathrm{FV}_n^m(e_1) \cup \mathrm{FV}_n^m(e_2) \\
\mathrm{FV}_n^m(\uparrow e) &= \mathrm{FV}_n^{m+1}(e) \\
\mathrm{FV}_n^{m+1}(\downarrow e) &= \mathrm{FV}_n^m(e) \\
\mathrm{FV}_n^m(b) &= \{\,\} \\
\mathrm{FV}_n^m(t_1 \to t_2) &= \mathrm{FV}_n^m(t_1) \cup \mathrm{FV}_n^m(t_2) \\
\mathrm{FV}_n^m([\gamma]t) &= \mathrm{FV}_n^{m+1}(\gamma) \cup \mathrm{FV}_n^{m+1}(t) \\
\mathrm{FV}_n^m(\emptyset) &= \{\,\} \\
\mathrm{FV}_n^m(\gamma_1, \gamma_2) &= \mathrm{FV}_n^m(\gamma_1) \cup \mathrm{FV}_n^m(\gamma_2) \\
\mathrm{FV}_m^m(x\!:\!t) &= \{x\} \\
\mathrm{FV}_n^m(x\!:\!t) &= \{\,\}, \quad \textit{if } m \neq n
\end{aligned}
$$

The definition of free variables induces a notion of α-equivalence. $\lambda^{[]}$ then adheres to the following conventions. We occasionally state Convention 2 as an explicit side condition.

Convention 1 *α-equivalent terms (or types) are interchangeable in all contexts.*

Convention 2 (Barendregt [2]) *Bound and free variables are assumed to be different. (If necessary, Convention 1 can be used to rename the bound ones.)*

Definition 2 (Substitution). *The result of the capture-avoiding substitution of the stage-0 term e' for the stage-0 variable x in the stage-m term e is defined by*

$$
\begin{aligned}
(c_{\{t\}})^m\{e'/x\} &= c_{\{t\}} \\
(x)^0\{e'/x\} &= e' \\
(x)^m\{e'/x\} &= x, \qquad \textit{if } m \neq 0 \\
(\lambda y\!:\!t.\, e)^m\{e'/x\} &= \lambda y\!:\!t.\, (e)^m\{e'/x\} \\
(e_1\, e_2)^m\{e'/x\} &= (e_1)^m\{e'/x\}\, (e_2)^m\{e'/x\} \\
(\uparrow e)^m\{e'/x\} &= \uparrow (e)^{m+1}\{e'/x\} \\
(\downarrow e)^{m+1}\{e'/x\} &= \downarrow (e)^m\{e'/x\}
\end{aligned}
$$

Notice that by Convention 2, x and y are different in the rule for substitution under lambdas.

The operational semantics of $\lambda^{[]}$ is presented in Sect. 3.4.

2.2 Properties of $\lambda^{[]}$

The type system of $\lambda^{[]}$ in Fig. 1 is sound with respect to a standard hygienic semantics of staging primitives. (The proof of soundness is outlined in Sect. 4.)

$\lambda^{[]}$ supports both a function run for immediate evaluation of code values and mutable state. We defer the detailed treatment of these to Sect. 3 and give just an outline here. We assume the existence of type-indexed families of constant symbols $\mathsf{run}_{\{[]t\to t\}}$, $\mathsf{ref}_{\{t\to t\,\mathsf{ref}\}}$, $\mathsf{get}_{\{t\,\mathsf{ref}\to t\}}$, and $\mathsf{set}_{\{t\,\mathsf{ref}\to t\to t\}}$. Given their usual semantics, these operations are type safe.

2.3 Examples

An implementation in $\lambda^{[]}$ of the classic staged power function is shown below. (In this implementation, we have taken the liberty to use standard integer arithmetic, monomorphic recursive let-expressions, and conditionals. They can all be added straightforwardly to $\lambda^{[]}$.)

$$
\begin{array}{l}
\mathsf{pow_{gen}} : \mathsf{int} \to [\,](\mathsf{int} \to \mathsf{int}) = \\
\quad \lambda n : \mathsf{int}. \\
\quad\quad \uparrow(\lambda x : \mathsf{int}. \\
\quad\quad\quad \downarrow(\mathsf{let\ rec\ pow_{bt}} : \mathsf{int} \to [x{:}\mathsf{int}]\mathsf{int} \to [x{:}\mathsf{int}]\mathsf{int} = \\
\quad\quad\quad\quad\quad \lambda n : \mathsf{int}.\, \lambda c : [x{:}\mathsf{int}]\mathsf{int}. \\
\quad\quad\quad\quad\quad\quad \mathsf{if}\ n = 0\ \mathsf{then} \uparrow 1\ \mathsf{else} \uparrow(\downarrow c \times \downarrow(\mathsf{pow_{bt}}\,(n-1)\,c)) \\
\quad\quad\quad\quad \mathsf{in\ pow_{bt}}\ n \uparrow x))
\end{array}
$$

This example demonstrates the typical use of staged computation to implemented partial evaluation: The staging primitives are used to define a *binding-time separated* function $\mathsf{pow_{bt}}$ of type $\mathsf{int} \to [x : \mathsf{int}]\mathsf{int} \to [x : \mathsf{int}]\mathsf{int}$ and a *generating extension* $\mathsf{pow_{gen}}$ of type $\mathsf{int} \to [](\mathsf{int} \to \mathsf{int})$. Evaluating $\mathsf{pow_{gen}}\,n$ yields the textual representation of a function of type $\mathsf{int} \to \mathsf{int}$ that computes x^n. For example, $\mathsf{pow_{gen}}\,3$ yields $\uparrow(\lambda x : \mathsf{int}.\, x \times x \times x \times 1)$ and $\mathsf{run}_{\{[](\mathsf{int}\to\mathsf{int})\to\mathsf{int}\to\mathsf{int}\}}\,(\mathsf{pow_{gen}}\,3)$ yields a function of type $\mathsf{int} \to \mathsf{int}$ that computes x^3.

The following shows an example where a code value ($\uparrow x$) with a free variable (x) is passed across the future-stage λ of another variable (y) by the means of an assignment. This example is well typed in $\lambda^{[]}$ since the free variable x does not escape its scope. (To ease readability, the types of the constants have been left out from this example.)

$$
\begin{array}{l}
\uparrow(\lambda x : \mathsf{int}. \\
\quad \downarrow(\mathsf{let}\, c : [x : \mathsf{int}]\mathsf{int} = \mathsf{ref}_{\{\dots\}}\,(\uparrow 1)\,\mathsf{in} \uparrow(\lambda y : t. \downarrow(\,\mathsf{set}_{\{\dots\}}\ c\ (\uparrow x)\ ;\ \cdots\ ))))
\end{array}
$$

On the other hand, the following classic attempt to pass a future-stage variable beyond its own scope is correctly rejected by the type system.

$$
\mathsf{let}\ c : t = \mathsf{ref}_{\{\dots\}}\ (\uparrow 1)\ \mathsf{in} \uparrow(\lambda x : \mathsf{int}. \downarrow(\,\mathsf{set}_{\{\dots\}}\ c\ (\uparrow x); \cdots\ ))
$$

In this example, the type of $\uparrow x$ is $[x : \mathsf{int}]\mathsf{int}$. Hence the type t of c must be $([x : \mathsf{int}]\mathsf{int})\,\mathsf{ref}$. But at the let-binding of c, this type is not well kinded since it refers to a unbound (future-stage) variable x.

3 The Staged Type System $\lambda^{[]}_{<:}$

As a type system for staged computation, $\lambda^{[]}$ is strikingly concise (as witnessed by its definition in Fig. 1). But $\lambda^{[]}$ lacks the expressive power required by general-purpose staged programming languages.

In this section, we first identify two examples representing lack of expressiveness in $\lambda^{[]}$. We argue that *subtype polymorphism* provide the expressiveness necessary to handle these examples. We therefore extend $\lambda^{[]}$ with a subtyping fragment, in Sect. 3.1. We then define the evaluation fragment and the imperative fragment of $\lambda^{[]}_{<:}$, in Secs. 3.2 and 3.3. All fragments provide orthogonal features that can be added to $\lambda^{[]}$ independently of each other. Finally, in Sect. 3.4, we define the operational semantics of $\lambda^{[]}_{<:}$.

Shortcoming 1. Consider the following term, which generates a closed code value and then splices that value into a context containing a variable x.

$$\mathsf{let}\ c = \uparrow 1\ \mathsf{in}\ \uparrow(\lambda x : t.\downarrow c)$$

This term in not typable in $\lambda^{[]}$: The typing rule for $\downarrow$ insists that the current type environment (in this example the one that declares x) must be identical to the type environment of the code spliced in (in this example an empty one). In $\lambda^{[]}$, a code value of type $[\gamma]t$ can only be spliced into a context that provides *exactly* the binding in γ. But recall the intuition that, if c has type $[\gamma]t$ then it denotes a representation of a term that has type t in type environment γ. Then it seems clear that c also has type $[\gamma']t$ for an extended environment γ' of γ. Similar reasoning is found within type systems for object calculi, record types, and other type systems with a built-in notion of subsumption.

To address this shortcoming, $\lambda^{[]}_{<:}$ therefore extends $\lambda^{[]}$ with a rule of subsumption that allows us to *weaken* a code type (such as the type of $\downarrow c$ in the example above) by adding unused bindings to its environment. This idea is formalized in Sect. 3.1 below.

Shortcoming 2. Consider the following term that invokes run on a closed code value under a context containing a variable x: (The term is incomplete; we assume that the run is at stage 0.)

$$\uparrow(\lambda x : t.\downarrow(\cdots \mathsf{run}_{\{\cdots\}}\,(\uparrow 1)\cdots))$$

This term in not typable $\lambda^{[]}$: The $\uparrow 1$ has type $[x : t]\mathsf{int}$ but run expects a closed code value of type $[]\mathsf{int}$. The typing rule for $\uparrow$ insists that the type environment of the code generated is identical to the type environment used to type the future-stage term under the $\uparrow$ (in this example the one that declares x). But intuition tells us that if x is not used in the body of e, then the type of $\uparrow e$ need not mention x.

To address this shortcoming, $\lambda^{[]}_{<:}$ also extends $\lambda^{[]}$ with a rule of subsumption that allows us to *strengthen* a typing context (such as the one used when typing

Additional Typing rules: $\boxed{\Gamma \,;\, \Gamma' \vdash e : t}$

$$\frac{\Gamma_1 \,;\, \Gamma_1' \vdash \Gamma_2 \,;\, \Gamma_2' \quad \Gamma_1 \,;\, \Gamma_1' \vdash t_1 : * \quad \Gamma_2 \,;\, \Gamma_2' \vdash e : t_2 \quad \Gamma_1 \,;\, \Gamma_1' \leq \Gamma_2 \,;\, \Gamma_2' \quad t_2 \leq t_1}{\Gamma_1 \,;\, \Gamma_1' \vdash e : t_1} \ (\leq)$$

Subtyping rules: $\boxed{t \leq t'}$

$$\frac{}{t \leq t} \ (\text{S-Refl}) \qquad \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3} \ (\text{S-Trans})$$

$$\frac{t_1 \leq t_2}{\forall T :: \kappa.\, t_1 \leq \forall T :: \kappa.\, t_2} \ (\text{S-All}) \qquad \frac{t_2 \leq t_1 \quad t_1' \leq t_2'}{t_1 \to t_1' \leq t_2 \to t_2'} \ (\text{S-Arrow})$$

$$\frac{t_1 \leq t_2}{\uparrow t_1 \leq \uparrow t_2} \ (\text{S-Up}) \qquad \frac{\gamma_2 \leq \gamma_1 \quad t_1 \leq t_2}{[\gamma_1]t_1 \leq [\gamma_2]t_2} \ (\text{S-Box})$$

$$\frac{t_1 \leq t_2}{\downarrow t_1 \leq \downarrow t_2} \ (\text{S-Down}) \qquad \frac{t_1 \leq t_2}{x : t_1 \leq x : t_2} \ (\text{S-Bind})$$

$$\frac{}{\gamma \leq \emptyset} \ (\text{S-Width}) \qquad \frac{\gamma_1 \leq \gamma_1' \quad \gamma_2 \leq \gamma_2'}{\gamma_1, \gamma_2 \leq \gamma_1', \gamma_2'} \ (\text{S-JoinCongr})$$

$$\frac{}{\gamma_1, (\gamma_2, \gamma_3) \leq (\gamma_1, \gamma_2), \gamma_3} \ (\text{S-JoinAssoc}) \qquad \frac{}{\gamma_1, \gamma_2 \leq \gamma_2, \gamma_1} \ (\text{S-JoinComm})$$

Fig. 2. The Subtype Fragment of $\lambda^{[]}_{\leq}$

$\uparrow 1$ in the example above) by removing unused bindings from its environments. This idea is also formalized in Sect. 3.1 below.

In the rest of this article, we let $\mathbf{V}$ and $\mathbf{L}$ denote disjoint sets of term variables and store locations, respectively. We let x and y range over $\mathbf{V}$ and ℓ over $\mathbf{L}$ (with primes and subscripts applied when necessary). We let ξ range over $\mathbf{V} \cup \mathbf{L}$.

3.1 Subtype Polymorphism

Figure 2 contains the subtype fragment of $\lambda^{[]}_{\leq}$. The subtype rules follows standard subtype rules for records and objects [20]. Notice that the code type is contravariant in the type environment and covariant in the type. The subsumption rule states that if we can type e with type environment γ, then we can also type it with type environment that provides more variable bindings than γ. This rule uses a pointwise extension of $\leq$ to sequences of type environments and the following iterative well kindedness rule for sequences of environments.

$$\frac{}{\,;\, \Gamma_1' \vdash \Gamma_2} \qquad \frac{\Gamma_1 \,;\, \gamma_1 \cdot \Gamma_1' \vdash \gamma_2 \quad \Gamma_1 \,;\, \gamma_1 \cdot \Gamma_1' \vdash \Gamma_2}{\Gamma_1 \cdot \gamma_1 \,;\, \Gamma_1' \vdash \Gamma_2 \cdot \gamma_2}$$

Lemma 1. *If* $\Gamma \,;\, \Gamma' \vdash t :: *$ *and* $t \leq t'$ *then also* $\Gamma \,;\, \Gamma' \vdash t' :: *$

Additional syntax:

$$\begin{aligned}
&\text{(Terms) } e ::= \cdots \mid \rho\theta.\, e \\
&\text{(Types) } t ::= \cdots \mid t\ \mathsf{ref} \\
&\text{(Stores) } \theta \ = \ \{\langle \ell_i, v_i^0 \rangle \mid 1 \le i \le k\}
\end{aligned}$$

Additional typing rules: $\boxed{\Gamma\ ;\ \Gamma' \vdash e : t}$

$$\frac{\begin{array}{c}(\gamma, \gamma')\ ;\ \Gamma \cdot \Gamma' \vdash v_i^0 : t_i \quad 1 \le i \le k \\ (\gamma, \gamma') \cdot \Gamma\ ;\ \Gamma' \vdash e : t\end{array}}{\begin{array}{c}\gamma, \Gamma\ ;\ \Gamma' \vdash \rho\{\langle \ell_1, v_1^0 \rangle, \ldots, \langle \ell_k, v_k^0 \rangle\}.\, e : t \\ \text{where } \gamma' = (\ell_1 : t_1\ \mathsf{ref}, \ldots, \ell_k : t_k\ \mathsf{ref})\end{array}} \quad (\rho)$$

Fig. 3. The imperative fragment of $\lambda^{[]}_{<}$

3.2 Evaluation and Lifting

The evaluation and lifting fragment of $\lambda^{[]}_{<}$ introduces two (type-indexed families of) constants, $\mathsf{run}_{\{[]t \to t\}}$ and $\mathsf{lift}_{\{t \to []t\}}$, that serve dual purposes: run maps a future-stage value "back" to the present stage (called *demotion*) while lift maps a present-stage term "forward" to a future stage (called *promotion*).

The type of run guarantees that the argument is a *closed* future-stage term. It prevents, for example, the reduction $\mathsf{run}(\uparrow x) \longmapsto x$ in which a future-stage x percolates into the present stage. (See the reduction rule for run in Fig. 5 below.)

3.3 Mutable State

Figure 3 presents the type system for the imperative fragment of $\lambda^{[]}_{<}$. Mutable state is modeled syntactically using the approach pioneered by Felleisen and Hieb [11] and further developed by Wright and Felleisen [28]. Stores θ are finite sets of pairs containing cells and stage-0 values. As in Wright and Felleisen's work, we introduce a separate class of evaluation context, $\mathcal{R}$, that order the imperative operations according to left-to-right, call-by-value evaluation. Figure 4 below defines all evaluation contexts.

The syntactic approach to mutable state lets us express garbage collection on a store. Wright and Felleisen do not need to consider garbage collection, but the rule that we present is standard [17]. It is necessary in the proof of Progress (Lemma 11) for future-stage lambdas that contain ρ-terms. Here, garbage collection allows the reduction $\lambda x : t.\, \rho\theta.\, v^{n+1} \longmapsto \lambda x : t.\, v^{n+1}$, since the stage-$n+1$ value v^{n+1} cannot (by definition) contain cells ℓ.

The typing rule of ρ-terms is similar to that of Wright and Felleisen. Cells are treated as stage-0 variables and hence must be added to the left-most type environment in the type judgment. We use the usual invariant subtype rule for the type of references [20]:

$$\frac{t_1 \leq t_2 \quad t_2 \leq t_1}{t_1\ \mathsf{ref} \leq t_2\ \mathsf{ref}} \qquad \text{(S-Ref)}$$

We use the following extensions of the definition of free variables and substitution for ρ-terms.

$$\begin{aligned}
\mathrm{FV}^m_m(\rho\theta.\, e) &= \mathrm{FV}^m_m(\mathrm{codom}\,\theta) \cup \left(\mathrm{FV}^m_m(e) - \mathrm{dom}\,\theta\right) \\
\mathrm{FV}^n_m(\rho\theta.\, e) &= \mathrm{FV}^n_m(\mathrm{codom}\,\theta) \cup \mathrm{FV}^n_m(e), \qquad \text{if } m \neq n \\
(\rho\theta.\, e)^m\{e'/x\} &= \rho\theta.\, (e)^m\{e'/x\}
\end{aligned}$$

Notice that since the store contains stage-0 values only, it cannot contain free variables.

Figure 3 explicitly restricts reduction to situations that do not result in future-stage variables leaving their scope. For this purpose, we let $\mathrm{BV}^m_n(\mathcal{R}^m_{n'})$ denote the set of bound stage-n variables in the stage-m context $\mathcal{R}^m_{n'}$.

Definition 3 (Bound variables). $\mathrm{BV}^m_n(\mathcal{R}^m_{n'})$ *denotes the stage-n variables in* $\mathcal{R}^m_{n'}$ *bound by* λ*s that "surround" the hole* $\Box$*, defined as follows.*

$$\begin{aligned}
\mathrm{BV}^m_n(\Box) &= \{\,\} \\
\mathrm{BV}^m_n(\mathcal{R}^m_{n'}\, e) &= \mathrm{BV}^m_n(\mathcal{R}^m_{n'}) \\
\mathrm{BV}^m_n(v^m\, \mathcal{R}^m_{n'}) &= \mathrm{BV}^m_n(\mathcal{R}^m_{n'}) \\
\mathrm{BV}^m_n(\uparrow\mathcal{R}^{m+1}_{n'}) &= \mathrm{BV}^{m+1}_n(\mathcal{R}^{m+1}_{n'}) \\
\mathrm{BV}^{m+1}_n(\downarrow\mathcal{R}^m_{n'}) &= \mathrm{BV}^m_n(\mathcal{R}^m_{n'}) \\
\mathrm{BV}^{m+1}_{m+1}(\lambda x\!:\!t.\, \mathcal{R}^{m+1}_{n'}) &= \{x\} \cup \mathrm{BV}^{m+1}_{m+1}(\mathcal{R}^{m+1}_{n'}) \\
\mathrm{BV}^{m+1}_n(\lambda x\!:\!t.\, \mathcal{R}^{m+1}_{n'}) &= \mathrm{BV}^{m+1}_n(\mathcal{R}^{m+1}_{n'}), \qquad \textit{if } m+1 \neq n
\end{aligned}$$

In the definition of bound variables, m denotes the stage of the context while n denotes the stage of the hole in that context.

3.4 Semantics of $\boldsymbol{\lambda^{[]}_{<}}$

The reduction semantics of $\lambda^{[]}_{<}$ is defined by Figs. 4 and 5. Notice that the definition of values and contexts actually defines stage-indexed families of inductive terms. As remarked by Taha [23], the style of inductive definition is slightly unusual because it actually involves an infinite number of meta-variables. However, the values and contexts defined by these rules are still finite and admits inductive reasoning.

In the reduction rules in Fig. 5, $\oplus$ denotes a disjoint union.

4 Formal Properties

In this section we outline a formal proof of the soundness of the type system $\lambda^{[]}_{<}$ with respect to its reduction semantics. The proof follows the standard approach

$$
\begin{array}{rrl}
\text{(Values)} & v^0 & ::= \lambda x\!:\!t.\, e \mid c_{\{t\}} \mid \ell \\
& v^m & ::= \uparrow v^{m+1} \\
& v^{m+1} & ::= x \mid \lambda x\!:\!t.\, v^{m+1} \mid v^{m+1}\, v^{m+1} \\
& v^{m+2} & ::= \downarrow v^{m+1} \\
(\mathcal{E}\text{-contexts}) & \mathcal{E}^m_m & ::= \Box \\
& \mathcal{E}^m_n & ::= \mathcal{E}^m_n\, e \mid v^m\, \mathcal{E}^m_n \mid \uparrow \mathcal{E}^{m+1}_n \mid \rho\theta.\, \mathcal{E}^m_n \\
& \mathcal{E}^{m+1}_n & ::= \downarrow \mathcal{E}^m_n \mid \lambda x\!:\!t.\, \mathcal{E}^{m+1}_n \\
(\mathcal{S}\text{-contexts}) & \mathcal{S}^m_m & ::= \Box \\
& \mathcal{S}^m_n & ::= \lambda x\!:\!t.\, \mathcal{S}^m_n \mid \mathcal{S}^m_n\, e \mid v^{m+1}\, \mathcal{S}^m_n \mid \uparrow \mathcal{S}^{m+1}_n \\
& \mathcal{S}^{m+1}_n & ::= \downarrow \mathcal{S}^m_n \\
(\mathcal{R}\text{-contexts}) & \mathcal{R}^m_n & ::= \Box \mid \mathcal{R}^m_n\, e \mid v^m\, \mathcal{R}^m_n \mid \uparrow \mathcal{R}^{m+1}_n \\
& \mathcal{R}^{m+1}_n & ::= \downarrow \mathcal{R}^m_n \mid \lambda x\!:\!t.\, \mathcal{R}^{m+1}_n
\end{array}
$$

Fig. 4. Values and evaluation contexts of $\lambda^{[]}_{<}$

to soundness using a reduction semantics [28], but we need additional core results do deal with open terms. In particular, we need to deal with bound variables in addition to free variables, strengthening in addition to weakening, and with demotion and promotion.

4.1 Standard Results

Lemma 2 (Decomposition). *Let* $\mathrm{dom}_n(\gamma_0 \cdot \cdots \cdot \gamma_m) = \mathrm{dom}(\gamma_n)$.

(a) If D_0 *is a derivation of* $\gamma \cdot \Gamma\ ;\ \Gamma' \vdash \mathcal{E}_0^{|\Gamma|}[e_1] : t$ *then there exists an environment* γ_1 *with* $\mathrm{dom}(\gamma_1) \subseteq \mathbf{L}$*, a stack* Γ_1'*, a type* t_1*, and a derivation* D_1 *of* $(\gamma, \gamma_1)\ ;\ \Gamma_1' \vdash e_1 : t_1$ *such that* D_1 *appears in* D_0 *at the location corresponding to the hole* $\Box$ *in* $\mathcal{E}_0^{|\Gamma|}$.

(b) If D_0 *is a derivation of* $\gamma \cdot \gamma' \cdot \Gamma\ ;\ \Gamma' \vdash \mathcal{S}_0^{|\Gamma|}[e_1] : t$ *then there exists an environment* γ_1'*, a stack* Γ_1'*, a type* t_1*, and a derivation* D_1 *of* $\gamma \cdot \gamma_1'\ ;\ \Gamma_1' \vdash e_1 : t_1$ *such that* D_1 *appears in* D_0 *at the location corresponding to the hole* $\Box$ *in* $\mathcal{S}_0^0$.

(c) If D_0 *is a derivation of* $\gamma \cdot \Gamma\ ;\ \Gamma' \vdash \mathcal{R}_0^{|\Gamma|}[e_1] : t$ *then there exists a stack* $\Gamma_1' \supseteq \Gamma'$*, a type* t_1*, and a derivation* D_1 *of* $\gamma\ ;\ \Gamma_1' \vdash e_1 : t_1$ *such that* D_1 *appears in* D_0 *at the location corresponding to the hole* $\Box$ *in* $\mathcal{R}_0^{|\Gamma|}$ *and for all* n, $\mathrm{dom}_n(\Gamma_1') - \mathrm{dom}_n(\Gamma') \subseteq \mathrm{BV}_n^0(\mathcal{R}_0^0)$.

The final part of case (c) states that the extra bindings in Γ_1' required to type e_1 can be traced back to bindings in the context. This is used in the proof of Subject Reduction for case (Set), where it guarantees that the value to store can be typed using the outer type environment of the surrounding ρ-expression, which potentially provides fewer bindings.

The dual notion of decomposition is replacement:

Reduction rules: $\boxed{e \longmapsto e'}$

$$\frac{e \longmapsto e'}{\mathcal{E}_0^0[e] \longmapsto \mathcal{E}_0^0[e']} \quad (\textsc{Ctx})$$

$$(\lambda x\!:\!t.\, e)\, v^0 \longmapsto (e)^0\{v^0/x\} \quad (\beta)$$

$$\uparrow\mathcal{S}_0^0[\downarrow\uparrow v^1] \longmapsto \uparrow\mathcal{S}_0^0[v^1] \quad (\downarrow\uparrow)$$

$$\mathsf{run}_{\{t\}}\,(\uparrow v^1) \longmapsto v^1 \quad (\textsc{Run})$$

$$\mathsf{lift}_{\{t\}}\, v^0 \longmapsto \uparrow v^0 \quad (\textsc{Lift})$$

$$\mathsf{ref}_{\{t\}}\, v^0 \longmapsto \rho\{\langle \ell, v^0\rangle\}.\, \ell \quad (\textsc{Ref})$$

$$\rho\{\langle \ell, v^0\rangle\} \oplus \theta.\, \mathcal{R}_0^0[\mathsf{get}_{\{t\}}\, \ell] \longmapsto \rho\{\langle \ell, v^0\rangle\} \oplus \theta.\, \mathcal{R}_0^0[v^0] \quad (\textsc{Get})$$
$$\text{if } \mathrm{BV}_n^0(\mathcal{R}_0^0) \cap \mathrm{FV}_n^0(v^0) = \{\,\}$$

$$\rho\{\langle \ell, v_1^0\rangle\} \oplus \theta.\, \mathcal{R}_0^0[\mathsf{set}_{\{t\}}\, \ell\, v_2^0] \longmapsto \rho\{\langle \ell, v_2^0\rangle\} \oplus \theta.\, \mathcal{R}_0^0[v_2^0] \quad (\textsc{Set})$$
$$\text{if } \mathrm{BV}_n^0(\mathcal{R}_0^0) \cap \mathrm{FV}_n^0(v_2^0) = \{\,\}$$

$$\rho\theta_1.\, \rho\theta_2.\, e \longmapsto \rho\theta_1 \oplus \theta_2.\, e \quad (\textsc{Merge})$$

$$\mathcal{R}_0^0[\rho\theta.\, e] \longmapsto \rho\theta.\, \mathcal{R}_0^0[e] \quad (\rho\text{-}\textsc{Lift})$$
$$\text{if } \mathrm{BV}_n^0(\mathcal{R}_0^0) \cap \mathrm{FV}_n(\theta) = \{\,\} \text{ and } \mathcal{R}_0^0 \neq \Box$$

$$\rho\theta_1 \oplus \theta_2.\, e \longmapsto \rho\theta_1.\, e \quad (\mathrm{GC})$$
$$\text{if } \theta_2 \neq \{\,\} \text{ and } \mathrm{dom}(\theta_2) \cap \mathrm{FV}_0(\theta_1) = \{\,\}$$
$$\text{and } \mathrm{dom}(\theta_2) \cap \mathrm{FV}_0^0(e) = \{\,\}$$

Evaluation:

$$\mathrm{eval}(e) = v^0, \quad \text{if } e \longmapsto^* v^0$$

Convention 2 applies to case (Ref), where it forces ℓ not to occur in v^0.

The side conditions of rules (Get) and (Set) prevent free variables in the contractum from being captured by the context. The side condition of (ρ-Lift) prevents variables bound in the context from escaping their scope. These side conditions are explicit instances of Convention 2.

The rule (GC) allows part of the store to be *garbage collected*, if its cells are not referred to.

Fig. 5. Reduction semantics of $\lambda_{<}^{[]}$

Lemma 3 (Replacement).

(a) If D_0 is a derivation of $\gamma \cdot \Gamma\ ;\ \Gamma' \vdash \mathcal{E}_0^{|\Gamma|}[e_1] : t$, D_1 is a derivation of $\gamma_1\ ;\ \Gamma_1' \vdash e_1 : t_1$, D_1 appears in D_0 at the location corresponding to the hole, and $\gamma_1\ ;\ \Gamma_1' \vdash e_1' : t_1$, then $\gamma \cdot \Gamma\ ;\ \Gamma' \vdash \mathcal{E}_0^{|\Gamma|}[e_1'] : t$.

(b) If D_0 is a derivation of $\gamma \cdot \gamma' \cdot \Gamma\,;\Gamma' \vdash \mathcal{S}_0^{|\Gamma|}[e_1] : t$, D_1 is a derivation of $\gamma_1\,;\Gamma_1' \vdash e_1 : t_1$, D_1 appears in D_0 at the location corresponding to the hole, and $\gamma_1\ ;\ \Gamma_1' \vdash e_1' : t_1$, then $\gamma \cdot \gamma' \cdot \Gamma\ ;\ \Gamma' \vdash \mathcal{S}_0^{|\Gamma|}[e_1'] : t$.

(c) If D_0 *is a derivation of* $\gamma \cdot \Gamma \,;\, \Gamma' \vdash \mathcal{R}_0^{|\Gamma|}[e_1] : t$, D_1 *is a derivation of* $\gamma_1 \,;\, \Gamma_1' \vdash e_1 : t_1$, D_1 *appears in* D_0 *at the location corresponding to the hole, and* $\gamma_1 \,;\, \Gamma_1' \vdash e_1' : t_1$, *then* $\gamma \cdot \Gamma \,;\, \Gamma' \vdash \mathcal{R}_0^{|\Gamma|}[e_1'] : t$.

Notice that we only ever substitute away stage-0 term variables (for a stage-0 values) in the evaluation rules.

Lemma 4 (Weakening). *If* $\gamma_0 \cdot \cdots \cdot \gamma_m \,;\, \gamma_{m+1} \cdot \cdots \cdot \gamma_k \vdash e : t$ *and for each* $0 \le i \le k$, $\gamma_i \subseteq \gamma_i'$ *and for each* $\xi \in \mathrm{dom}(\gamma_i') - \mathrm{dom}(\gamma_i)$, $\xi \notin \mathrm{FV}_i^m(e)$ *then* $\gamma_0' \cdot \cdots \cdot \gamma_m' \,;\, \gamma_{m+1}' \cdot \cdots \cdot \gamma_k' \vdash e : t$.

Lemma 5 (Strengthening). *If* $\gamma_0' \cdot \cdots \cdot \gamma_m' \,;\, \gamma_{m+1}' \cdot \cdots \cdot \gamma_k' \vdash e : t$ *and for each* $0 \le i \le k$, $\gamma_i \subseteq \gamma_i'$ *and for each* $\xi \in \mathrm{dom}(\gamma_i') - \mathrm{dom}(\gamma_i)$, $\xi \notin \mathrm{FV}_i^m(e)$ *then* $\gamma_0 \cdot \cdots \cdot \gamma_m \,;\, \gamma_{m+1} \cdot \cdots \cdot \gamma_k \vdash e : t$.

Strengthening is used in the proof of Subject Reduction, in the case (Set).

Lemma 6 (Substitution). *If* $(x : t', \gamma) \cdot \Gamma \,;\, \Gamma' \vdash e : t$ *and* $\gamma \,;\, \Gamma \cdot \Gamma' \vdash e' : t'$ *then* $\gamma \cdot \Gamma \,;\, \Gamma' \vdash (e)^{|\Gamma|}\{e'/x\} : t$.

4.2 Results for Staging

The following two lemmas show that demotion and promotion yield well-typed results: A well-typed value (but not necessarily an expression) at stage $m+1$ is also well-typed at stage m. And conversely, a well-typed value at stage m is also a well-typed value at stage $m+1$. These results are used in the proof of Subject Reduction (Lemma 9).

Lemma 7 (Demotion). *If* $\Gamma \cdot \gamma \cdot \emptyset \,;\, \Gamma' \vdash v^{|\Gamma|+1} : t$ *then* $\Gamma \cdot \gamma \,;\, \emptyset \cdot \Gamma' \vdash v^{|\Gamma|+1} : t$.

Lemma 8 (Promotion). *If* $\Gamma \cdot \gamma \,;\, \emptyset \cdot \Gamma' \vdash v^{|\Gamma|} : t$ *then* $\Gamma \cdot \gamma \cdot \emptyset \,;\, \Gamma' \vdash v^{|\Gamma|} : t$.

4.3 Subject Reduction

In the following statement of Subject Reduction, we need a general outer type environment γ (rather than an empty one) to account for cells in the store and we need the future-stage Γ' to account for future-stage variables.

Lemma 9 (Subject reduction). *If* $\gamma \,;\, \Gamma' \vdash e : t$ *and* $e \longmapsto e'$ *then* $\gamma \,;\, \Gamma' \vdash e' : t$.

Proof. by induction on $e \longmapsto e'$, with an induction case corresponding to reduction rule (Ctx), and with base cases for the remaining reduction rules. We make extensive use Decomposition (Lemma 2) and Replacement (Lemma 3). In each case, we apply a Typing Inversion lemma to deduce the types of subterms from the type of a term. This lemma is similar to that of Pierce [20, Sect. 15.3], but needs to account for stages. Substitution (Lemma 6) is used in case (β) and Demotion (Lemma 7) and Promotion (Lemma 8) are used in cases (Run) and (Lift).

4.4 Progress

We need the following definition of well-formed sequences of type environments.

Definition 4. *A sequence of type environments, $\Gamma\,;\Gamma'$, is well-formed if it satisfies the following rules.*

$$\vdash \epsilon\,;\Gamma' \qquad \frac{\Gamma\cdot\gamma\ ;\ \Gamma'\vdash\gamma \quad \vdash\Gamma\,;\gamma\cdot\Gamma'}{\vdash\Gamma\cdot\gamma\,;\Gamma'}$$

Lemma 10. *If $\vdash \Gamma\cdot\gamma\cdot\Gamma'\,;\epsilon$ and $\Gamma\cdot\gamma\ ;\ \Gamma'\vdash t :: *$ then $\vdash\Gamma\cdot(x:t,\gamma)\cdot\Gamma'\,;\epsilon$.*

Lemma 11 (Progress). *If $\emptyset\ ;\ \overline{\emptyset}\vdash e : t$ then either $e = v^0$, $e = \rho\theta.\,v^0$, or $e = \mathcal{E}_0^0[e']$ where $e'\longmapsto e''$. (Here $\overline{\emptyset}$ denotes a sufficiently long sequence of $\emptyset$s.)*

Proof. This follows as a corollary of a more general lemma that states that if $\gamma = (\ell_1 : t_1\,\mathsf{ref},\ldots,\ell_k : t_k\,\mathsf{ref})$ with $\gamma\ ;\ \Gamma\cdot\Gamma'\vdash t_i :: *$ for any $1\le i\le l$ and $\vdash\gamma\cdot\Gamma\cdot\Gamma'\,;\epsilon$ and $\gamma\cdot\ ;\ \Gamma'\vdash e : t$ then either

(a) $e = v^{|\Gamma|}$ for some stage-$|\Gamma|$ value $v^{|\Gamma|}$,
(b) $e = \rho\theta.\,v^{|\Gamma|}$ for some θ and stage-$|\Gamma|$ value $v^{|\Gamma|}$,
(c) $e = \mathcal{E}_0^{|\Gamma|}[e']$ where $e'\longmapsto e''$ for some context $\mathcal{E}_0^{|\Gamma|}$ and terms e' and e'',
(d) $e = \mathcal{R}_0^{|\Gamma|}[\mathsf{get}_{\{t'\}}\,\ell]$,
(e) $e = \rho\theta.\,\mathcal{R}_0^{|\Gamma|}[\mathsf{get}_{\{t'\}}\,\ell]$ where $\ell\notin\operatorname{dom}\theta$,
(f) $e = \mathcal{R}_0^{|\Gamma|}[\mathsf{set}_{\{t'\}}\,\ell\,v^0]$, or
(g) $e = \rho\theta.\,\mathcal{R}_0^{|\Gamma|}[\mathsf{set}_{\{t'\}}\,\ell\,v^0]$ where $\ell\notin\operatorname{dom}\theta$.

Together, Subject Reduction and Progress establish type soundness of $\lambda^{[]}_{<}$.

5 Relation with Existing Type Systems

We show that $\lambda^{[]}$ types at least the terms typable in Davies's $\lambda^{\bigcirc}$ and that $\lambda^{[]}_{<}$ types at least the terms typable in Davies and Pfennings's λ^{S4}. (In the following subsections, we abuse notation by using the same meta variable for syntactic categories in different calculi.)

The next and prev of $\lambda^{\bigcirc}$ directly match $\uparrow$ and $\downarrow$ of $\lambda^{[]}_{<}$. Conversely, the unbox_n of λ^{S4} serves two purposes, namely as an iterated $\downarrow$ and as run. Hence the translation of λ^{S4} into $\lambda^{[]}_{<}$ is slightly more involved than translation of $\lambda^{\bigcirc}$ into $\lambda^{[]}_{<}$.

5.1 Relation to $\lambda^{\bigcirc}$

$\lambda^{\bigcirc}$ extends the simply-typed λ-calculus as follows.

Definition 5 ($\lambda^{\bigcirc}$).

$$\begin{aligned}
(\textit{Types})\ \ & t ::= b \mid t_1 \to t_2 \mid \bigcirc t \\
(\textit{Terms})\ \ & e ::= x \mid \lambda x\!:\!t.\, e \mid e_1\, e_2 \mid \mathsf{next}\, e \mid \mathsf{prev}\, e \\
(\textit{Environments})\ & \Gamma ::= \emptyset \mid x : t^n, \Gamma
\end{aligned}$$

The typing judgment of $\lambda^{\bigcirc}$, $\Gamma \vdash^n e : t$, *is defined by the following typing rules.*

$$\frac{x : t^n \in \Gamma}{\Gamma \vdash^n x : t} \qquad \frac{\Gamma \vdash^{n+1} e : t}{\Gamma \vdash^n \mathsf{next}\, e : \bigcirc t} \qquad \frac{\Gamma \vdash^n e : \bigcirc t}{\Gamma \vdash^{n+1} \mathsf{prev}\, e : t}$$

$$\frac{x : t^n, \Gamma \vdash^n e : t'}{\Gamma \vdash^n \lambda x\!:\!t.\, e : t \to t'} \qquad \frac{\Gamma \vdash^n e_1 : t_2 \to t \quad \Gamma \vdash^n e_2 : t_2}{\Gamma \vdash^n e_1\, e_2 : t}$$

The typing judgments of $\lambda^{\bigcirc}$ carry one type environment Γ that tracks the stage of variables by means of an integer staging annotation, $x : t^n$. In $\lambda^{[]}$, this staging information is implicitly given by the index into the stack where a variable is found. Hence, we parametrize the translation from $\lambda^{\bigcirc}$ into $\lambda^{[]}$ by a $\lambda^{\bigcirc}$ type environment and an integer that denotes the current stage.

Definition 6 (Translating $\lambda^{\bigcirc}$ into $\lambda^{[]}$). *Given a* $\lambda^{\bigcirc}$ *typing context* Γ *and a non-negative integer* n, *we generate a* $\lambda^{[]}$ *type environment containing the stage-*n *variables in* Γ *by*

$$[\![\bigcirc]\!]^{\Gamma}_m = \left(x : [\![t]\!]^{\Gamma}_m \mid x : t^m \in \Gamma\right)$$

and we then translate $\lambda^{\bigcirc}$ *types and terms into* $\lambda^{[]}$ *types by*

$$\begin{aligned}
[\![b]\!]^{\Gamma}_m &= b & [\![x]\!]^{\Gamma}_m &= x \\
[\![t_1 \to t_2]\!]^{\Gamma}_m &= [\![t_1]\!]^{\Gamma}_m \to [\![t_2]\!]^{\Gamma}_m & [\![\lambda x\!:\!t.\, e]\!]^{\Gamma}_m &= \lambda x : [\![t]\!]^{\Gamma}_m .\, [\![e]\!]^{x:t^m,\Gamma}_m \\
[\![\bigcirc t]\!]^{\Gamma}_m &= [[\![\bigcirc]\!]^{\Gamma}_{m+1}][\![t]\!]^{\Gamma}_{m+1} & [\![e_1\, e_2]\!]^{\Gamma}_m &= [\![e_1]\!]^{\Gamma}_m\, [\![e_2]\!]^{\Gamma}_m \\
& & [\![\mathsf{next}\, e]\!]^{\Gamma}_m &= \uparrow [\![e]\!]^{\Gamma}_{m+1} \\
& & [\![\mathsf{prev}\, e]\!]^{\Gamma}_{m+1} &= \downarrow [\![e]\!]^{\Gamma}_m
\end{aligned}$$

Given a $\lambda^{\bigcirc}$ *typing context* Γ *and a pair of non-negative integers* m, n, *we generate a stack of* $\lambda^{[]}$ *type environment as follows.*

$$[\![\Gamma]\!]_{m,n} = \begin{cases} \epsilon, & \textit{if } m > n \\ [\![\bigcirc]\!]^{\Gamma}_m \cdot [\![\Gamma]\!]_{m+1,n}, & \textit{otherwise} \end{cases}$$

Finally, $\lambda^{\bigcirc}$ *type judgments are translated into* $\lambda^{[]}$ *as follows.*

$$\begin{aligned}
[\![\Gamma \vdash^m e : t]\!] = [\![\Gamma]\!]_{0,m}\ ;\ [\![\Gamma]\!]_{m+1,k} \vdash [\![e]\!]^{\Gamma}_m : [\![t]\!]^{\Gamma}_m \\
\textit{where } k = \max\{m \mid x : t^m \in \Gamma\}
\end{aligned}$$

This translation makes it evident that $\bigcirc t$ is a type of *open* code. The following lemma shows that the translation preserves typing. The proof follows by a straightforward induction on the derivation of the $\lambda^{\bigcirc}$ judgment using, in the case for abstractions, the observation that $[\![t]\!]_m^{x:t^m,\Gamma} = [\![t]\!]_m^{\Gamma}$ and hence also $[\![e]\!]_m^{x:t^m,\Gamma} = [\![e]\!]_m^{\Gamma}$.

Lemma 12. *If $\Gamma \vdash^m e : t$ is derivable in $\lambda^{\bigcirc}$ then $[\![\Gamma \vdash^m e : t]\!]$ is derivable in $\lambda^{[]}$.*

5.2 Relation to λ^{S4}

λ^{S4} is a variant of the modal type system $\lambda^{\Box}$ that replaces a combination of **unbox** with a number of **pop** operations by a single $\mathbf{unbox}_n$ operation [10].

Definition 7 (λ^{S4}).

$$\begin{aligned}
(\textit{Types})\ \ t &::= b \mid t_1 \to t_2 \mid \Box t \\
(\textit{Terms})\ \ e &::= x \mid \lambda x\!:\!t.\, e \mid e_1\, e_2 \mid \mathsf{box}\, e \mid \mathsf{unbox}_n\, e \\
(\textit{Environments})\ \Gamma &::= \emptyset \mid x : t, \Gamma \\
(\textit{Stacks})\ \Psi &::= \epsilon \mid \Psi; \Gamma
\end{aligned}$$

The typing judgment of λ^{S4}, $\Psi; \Gamma \vdash e : t$, is defined by the following typing rules.

$$\frac{x : t \in \Gamma}{\Psi; \Gamma \vdash x : t}$$

$$\frac{\Psi; \Gamma; \emptyset \vdash e : t}{\Psi; \Gamma \vdash \mathsf{box}\, e : \Box t} \qquad \frac{\Psi; \Gamma \vdash e : \Box t}{\Psi; \Gamma; \Gamma_1; \cdots; \Gamma_n \vdash \mathsf{unbox}_n\, e : t}$$

$$\frac{\Psi; (x : t, \Gamma) \vdash e : t'}{\Psi; \Gamma \vdash \lambda x\!:\!t.\, e : t \to t'} \qquad \frac{\Psi; \Gamma \vdash e_1 : t_2 \to t \quad \Psi; \Gamma \vdash e_2 : t_2}{\Psi; \Gamma \vdash e_1\, e_2 : t}$$

We simplify the translation into $\lambda^{[]}_{<:}$ by using constants run and lift that are not explicitly type annotated. (This can be avoided by defining the translation on typing judgments.) We also simplify lifting by assuming an implicit surrounding $\downarrow$. We thus write $\%e$ for $\downarrow(\mathsf{lift}\, e)$. More concretely, we use the following variants of the typing rules of run and lifting, both of which are admissible in $\lambda^{[]}_{<:}$. The rule for lifting follows by a use of subsumption to strengthen the γ' to $\emptyset$.

$$\frac{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash e : [\,]t}{\Gamma \cdot \gamma\ ;\ \Gamma' \vdash \mathsf{run}\, e : t} \qquad \frac{\Gamma \cdot \gamma\ ;\ \emptyset \cdot \Gamma' \vdash e : t}{\Gamma \cdot \gamma \cdot \gamma'\ ;\ \Gamma' \vdash \%\, e : t}$$

Definition 8 (Translating λ^{S4} into $\lambda^{[]}_{<}$). *We translate λ^{S4} types, terms, and judgments into $\lambda^{[]}_{<}$ by*

$$\begin{array}{rlrl} [\![b]\!] &= b & [\![x]\!] &= x \\ [\![t_1 \to t_2]\!] &= [\![t_1]\!] \to [\![t_2]\!] & [\![\lambda x\!:\!t.\, e]\!] &= \lambda x\!:\![\![t]\!].\, [\![e]\!] \\ [\![\Box t]\!] &= [\,][\![t]\!] & [\![e_1\, e_2]\!] &= [\![e_1]\!]\, [\![e_2]\!] \\ & & [\![\mathsf{box}\, e]\!] &= \uparrow [\![e]\!] \\ & & [\![\mathsf{unbox}_0\, e]\!] &= \mathsf{run}\, [\![e]\!] \\ & & [\![\mathsf{unbox}_{n+1}\, e]\!] &= \%^n \downarrow [\![e]\!] \end{array}$$

$$[\![\Psi; \Gamma \vdash e : t]\!] = [\![\Psi]\!] \cdot [\![\Gamma]\!] \;;\; \overline{\emptyset} \vdash [\![e]\!] : [\![t]\!]$$

This translation makes it evident that $\Box t$ is a type of *closed* code. The translation is similar to the translation of λ^{S4} into λ^{i}_{let} [6]. The following lemma shows that the $\lambda^{[]}_{<}$ variant of an unbox_n can remove the n topmost type environments. It follows by induction on n using the subsumption rule to discard the top-most assumptions.

Lemma 13. *If $\Gamma \cdot \gamma \;;\; \overline{\emptyset} \vdash e : [\,]t$ then $\Gamma \cdot \gamma \cdot \gamma_1 \cdot \cdots \cdot \gamma_{n+1} \;;\; \overline{\emptyset} \vdash \%^{n+1} \downarrow e : t$.*

We can then show that the translation of λ^{S4} into $\lambda^{[]}$ preserves typing. The proof follows by induction on the derivation of the typing judgment in λ^{S4}.

Lemma 14. *If $\Psi; \Gamma \vdash e : t$ is derivable in λ^{S4} then $[\![\Psi; \Gamma \vdash e : t]\!]$ is derivable in $\lambda^{[]}_{<}$ with the above typing rules for the* run *and* $\%$.

6 Related Work

Traditionally, type systems for staged programming are classified according to their ability to distinguish between open and closed code values. Davies's $\lambda^{\bigcirc}$ [9] and Davies and Pfenning's $\lambda^{\Box}$ [10] were the first type systems to introduce types of code values and the notion of multiple stages. The type system of $\lambda^{\bigcirc}$ cannot distinguish between open and closed code. Hence $\lambda^{\bigcirc}$ does not support an eval function.

The type system of MetaML is based on $\lambda^{\bigcirc}$ [26]. It provides a type of open code and hence cannot guarantee that only closed code is executed. Moggi et al. establish that guarantee by extending MetaML with an additional type of closed code [16]. Benaissa et al. generalize this type system by introducing a type that characterize closed *values*, not just closed *code values*, and then unifying the two code types into one of open code [4]. Neither of these type systems deal with mutable state. Calcagno et al. show that it is safe to store closed code values in reference cells [6].

The type system of $\lambda^{\Box}$ only allows closed code values [10]. An eval function can be encoded in $\lambda^{\Box}$. However, $\lambda^{\Box}$ does not support evaluation under future-stage λs. Hence, unlike approaches based on $\lambda^{\bigcirc}$, multi-stage programming in $\lambda^{\Box}$ leaves residual administrative redexes in code values.

Inspired by nominal logics, Nanevski proposes a variant of $\lambda^{\Box}$, called $\nu^{\Box}$, that allows code to contain free variables and that exposes these variables in the type of code [18]. This calculus introduces explicit substitutions to eliminate free variables. Nanevski et al. propose Contextual Modal Types in another variant of $\lambda^{\Box}$ with explicit substitutions [19]. Both of these calculi enable multi-stage programming that eliminates administrative redexes in code values.

Kim et al.'s λ^{sim}_{open} and λ^{poly}_{open} also extend $\lambda^{\Box}$ by parameterizing code types over type environments [14]. (Kim et al.'s type system generalizes a monomorphic type system previously presented by the present author [21].) But unlike the Contextual Modal Type system of Nanevski et al., λ^{sim}_{open} and λ^{poly}_{open} treats variables symbolically. This opens up for both a hygienic future-stage λ^* (via a capture-avoiding substitution) and a non-hygienic future-stage λ (via a capturing substitution). Unlike $\lambda^{[]}$, however, despite being hygienic λ^{sim}_{open} and λ^{poly}_{open} are not lexically scoped. For example, in these type systems a term such as $\uparrow(\lambda^* x : \mathsf{bool}.\downarrow(\mathsf{let}\ \_ = \uparrow(x+1)\ \mathsf{in}\ \cdots))$ is well typed despite the mismatching types of x.

Taha and Nielsen's λ^{α} combine open-code manipulation with an eval function by introducing explicit classifiers that name type environments [25]. The code type of λ^{α} is annotated by the classifier for the environment in which code can be inserted. A type-level quantifier over classifiers is used to make (code) types parametric in classifiers. In subsequent work, Calcagno et al. define the variant λ^{i}_{let} which simplify this idea by eliminating explicit classifiers in terms [7]. Neither of these calculi supports mutable state. We have addressed the warnings of Taha and Nielsen [25, Sect. 1.4] by demonstrating that α-equivalence is compatible with types that carry environments and that no negative side conditions on environments are required.

Chen and Xi's λ_{code} introduce a code type parameterized over a type environment and a type [8]. λ_{code} is nameless: variables are represented by their de Bruijn indices, both in terms and in types. Although the precise connection between $\lambda^{[]}$ and λ_{code} remains to be established, it seems that λ_{code} is similar to a nameless variant of $\lambda^{[]}$. Chen and Xi do not discuss mutable state.

Kameyama et al.'s $\lambda^{\oslash}_{1}$ extends a simple variant of λ^{α} with control effects [13]. $\lambda^{\oslash}_{1}$ supports mutable state and an eval function. To prevent scope extrusion, $\lambda^{\oslash}_{1}$ prohibits the evaluation of future-state λs to have observable side effects. In recent work, Westbrook et al. relax this requirement by characterizing terms as *weakly separable* when they do not have observable side effects that involve code values [27]. By requiring that escaped terms (i.e., the $\downarrow e$ of $\lambda^{[]}$) are weakly separable, Westbrook et al. can guarantee that no code value (and hence no future-stage variable) can leave the scope in which it is generated. In contrast (as witnessed by the example at the end of Sect. 2.3), $\lambda^{[]}$ allows both escaped terms and future-stage λs to have observable effects involving open code.

7 Conclusions

We have defined $\lambda^{[]}$, a core type system for staged computation that is sound and hygienic and that supports open-code manipulation, a first-class eval function,

and mutable state. We have also extended $\lambda^{[]}$ with subtype polymorphism. The result is $\lambda^{[]}_{<:}$, a type system for staged computation that is at least as expressive as existing type systems for staged computation, but strikingly simpler.

References

[1] Backus, J.W., Beeber, R.J., Best, S., Goldberg, R., Haibt, L.M., Herrick, H.L., Nelson, R.A., Sayre, D., Sheridan, P.B., Stern, H., Ziller, I., Hughes, R.A., Nutt, R.: The Fortran automatic coding system. In: Techniques for Reliability, Proceedings of the Western Joint Computer Conference, pp. 188–198 (1957)

[2] Barendregt, H.: The Lambda Calculus — Its Syntax and Semantics. North-Holland (1984)

[3] Bawden, A.: Quasiquotation in Lisp. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas (1999)

[4] Benaissa, Z.-E.-A., Moggi, E., Taha, W., Sheard, T.: Logical modalities and multi-stage programming. In: Proceedings of the Workshop on Intuitionistic Modal Logics and Applications, Trento, Italy (July 1999)

[5] Bondorf, A., Jones, N.D., Mogensen, T., Sestoft, P.: Binding time analysis and the taming of self-application. DIKU rapport, University of Copenhagen, Copenhagen, Denmark (1988)

[6] Calcagno, C., Moggi, E., Taha, W.: Closed Types as a Simple Approach to Safe Imperative Multi-stage Programming. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 25–36. Springer, Heidelberg (2000)

[7] Calcagno, C., Moggi, E., Taha, W.: ML-Like Inference for Classifiers. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 79–93. Springer, Heidelberg (2004)

[8] Chen, C., Xi, H.: Meta-programming through typeful code representation. Journal of Functional Programming 15(6), 797–835 (2005)

[9] Davies, R.: A temporal-logic approach to binding-time analysis. In: Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, pp. 184–195. IEEE Computer Society Press, New Brunswick (1996)

[10] Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM 48(3), 555–604 (2001)

[11] Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Technical Report Rice COMP TR89–100, Department of Computer Science, Rice University, Houston, Texas (June 1989)

[12] Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. International Series in Computer Science. Prentice-Hall (1993)

[13] Kameyama, Y., Kiselyov, O., Shan, C.C.: Shifting the stage: staging with delimited control. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 111–120. ACM, Savannah (2009)

[14] Kim, I.-S., Yi, K., Calcagno, C.: A polymorphic modal type system for lisp-like multi-staged languages. In: Proceedings of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages, pp. 257–268. ACM Press, Charleston (2006)

[15] McCarthy, J.: LISP 1.5 Programmer's Manual. The MIT Press, Cambridge (1962)

[16] Moggi, E., Taha, W., Benaissa, Z.-E.-A., Sheard, T.: An Idealized MetaML: Simpler, and More Expressive. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 193–207. Springer, Heidelberg (1999)

[17] Morrisett, G., Felleisen, M., Harper, R.: Abstract models of memory management. In: Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture, pp. 66–77. ACM Press, La Jolla (1995)
[18] Nanevski, A.: Meta-programming with names and necessity. In: Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming, pp. 206–217. ACM Press, Pittsburgh (2002)
[19] Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic 9(3), 1–49 (2008)
[20] Pierce, B.C.: Types and Programming Languages. The MIT Press, Cambridge (2002)
[21] Rhiger, M.: First-class open and closed code fragments. Trends in Functional Programming 6, 127–144 (2007), Intellect
[22] Taha, W.: Multi-Stage programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute of Science and Technology (1999)
[23] Taha, W.: A sound reduction semantics for untyped CBN multi-stage computations. or, the theory of MetaML is non-trivial. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. ACM Press, Boston (2000)
[24] Taha, W., Benaissa, Z.-E.-A., Sheard, T.: Multi-Stage Programming: Axiomatization and Type Safety. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 918–929. Springer, Heidelberg (1998)
[25] Taha, W., Nielsen, M.F.: Environment classifiers. In: Proceedings of the Thirtieth Annual ACM Symposium on Principles of Programming Languages, pp. 26–37. ACM Press, New Orleans (2003)
[26] Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 203–217. ACM Press, Amsterdam (1997)
[27] Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: Proceedings of the ACM SIGPLAN 2010 Conference on Programming Languages Design and Implementation, pp. 400–411. ACM Press, Toronto (2010)
[28] Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115, 38–94 (1994)