A Polymorphic Modal Type System for Lisp-Like Multi-Staged Languages *

Ik-Soon Kim
Seoul National University
iskim@ropas.snu.ac.kr

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

Cristiano Calcagno
Imperial College
ccris@doc.ic.ac.uk

Abstract

This article presents a polymorphic modal type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's staging constructs (the quasi-quotation system). The combination is meaningful because ML is a practical higher-order, impure, and typed language, while Lisp's quasi-quotation system has long evolved complying with the demands from multi-staged programming practices. Our type system supports open code, unrestricted operations on references, intentional variable-capturing substitution as well as capture-avoiding substitution, and lifting values into code, whose combination escaped all the previous systems.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Theory

Keywords Multi-staged languages, Type systems, Polymorphic types, Modal types, ML, Let-polymorphism, Quasi-quotation, Lisp, Scheme, Record type, Type inference

1. Introduction

Staged computation, which explicitly divides a computation into separate stages, is a unifying framework for the existing program-generation systems. Partial evaluation [12, 5], runtime code generation [9, 19, 15, 16], function inlining, and macro expansion [23, 10] are all instances of staged computation. The stage levels can be arbitrarily large, determined by the nesting depth of program generations: stage 0 is for conventional non-staged programs, and a program of stage 0 generates a program of stage 1 that generates a program of stage 2, and so on.

The key aspect of multi-staged languages is to have code templates (program fragments) as first-class objects. Code templates are freely passed, stored, composed with code of other stages, and executed when appropriate.

This article presents a polymorphic type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's multi-staged programming constructs. The combination is meaningful because ML is a practical higher-order, impure, and typed language, while Lisp has long evolved to comply with the demands from multi-staged programming practices. Lisp's staged programming features are all included in its so-called "quasi-quote" system. This system supports open code templates, imperative operations with code templates, intentional variable-capturing substitution (at the sacrifice of alpha-equivalence) as well as capture-avoiding substitution (as "gensym" does) of free variables in open code templates, and lifting values into code templates. Our type system supports all of these features, allowing a programmer both type safety as well as the expressiveness that has so far been only available using the quasi-quotation operators in Lisp (or Scheme).

Contributions Our contributions are as follows.

- We present a polymorphic type system for a higher-order multi-staged language that supports all features of Lisp's quasi-quote programming:
  - Open code: code with free variables can be constructed and composed without restrictions.
  - Imperative operations with open code: open code can be stored, dereferenced, and overwritten without restrictions.
  - Intentional variable-capturing substitution at stages > 0 ("unhygienic" macros): hence alpha-equivalence at stages > 0 (i.e., during code definitions and expansions) is not preserved. This sacrifice, which may be unacceptable to a purely functional language, is a feature that Lisp's quasi-quote programmers have long enjoyed for efficiency and programming convenience.
  - Capture-avoiding substitution at stages > 0 ("hygienic" macros [14]): the target language has an explicit new-name generation construct like Lisp's "gensym." Programmers use this construct to rename bound variables at runtime in order to avoid an unintentional variable-capture.
- Our type system conservatively extends ML with Lisp's quasi-quote system. ML's let-polymorphism with the value restriction is conservatively extended for imperative staged programs that handle open code templates as first-class objects. Also, ML's let-polymorphism is orthogonally combined with a record polymorphism to allow a single open code template in multiple environments.
- We present the type system's principal type inference algorithm.

* This work is partially supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by IT Leading R&D Support Project of Korea Ministry of Information and Communication, and by Microsoft Research Asia.

POPL'06 January 11–13, Charleston, South Carolina, USA.

We believe our type system is the only one supporting open code, unrestricted operations on references, both hygienic and unhygienic manipulation of code, and a type inference algorithm, whose combination escaped all the previous systems.

This difference comes from three choices in our type system design: each code template's type includes a type environment that constrains its free variables' types; a record type for type environment allows the polymorphic use of code templates; stages are represented by stacks of type environments.

Comparisons Our type system differs from existing works as follows.

- Davies and Pfenning's system [7, 8] does not support open code templates. Nanevski and Pfenning's system [17, 18] supports open code, but does not support more than two stages and has no type inference algorithm. Calcagno et al.'s systems [3, 2] does not support imperative operations for open code templates. The environment classifiers [25, 3] allow restricted open code templates whose free variables' binders should be lexically visible at the same stage level. Chen and Xi's system [4] does not support free *named* variables inside code templates, hence it does not support hygienic manipulation of code templates. Ancona and Moggi's system [1] supports open code and imperative operations, but it supports only two stages, does not support intentional variable-capturing substitution and has no type inference algorithm. Rhiger's system [22] supports open code, intentional variable-capturing substitution and imperative operations, but is monomorphic.
- For programs that are accepted by existing multi-staged type systems such as $\lambda^{\Box}$ [7, 8], $\lambda^{\bigcirc}$ [6], and λ^{i} [3] without its cross-stage persistence, there exist their semantics-preserving, translated versions that are accepted by our type system (See Section 6).
- Alpha-equivalence in our language is preserved only for stage 0 (non-staged) expressions; we enforce only closed code to be evaluated at stage 0. For higher stages alpha-equivalence is not preserved. If name change occurs in expressions of stages > 0, the result program's semantics becomes different from the original one.

  This non alpha-equivalence at stages > 0 (i.e., "unhygienic" manipulation of code) has been prevalent in macro programming, hence, we think worthwhile to support it in our type system.
- For alpha-equivalence at stages > 0 (i.e., for "hygienic" manipulation of code, or for staged programs that needs renaming at runtime), programmers can always choose to use the explicit new-name generation construct ("gensym" in Lisp, λ^* in our language). The construct $\lambda^* x.e$ uniquely renames bound variable x at runtime whenever a code is plugged inside e.

Examples Our type system's ideas and notable features in contrast to existing systems are illustrated with examples as follows. In examples, we use a mixed notation of Lisp's quasi-quote syntax [23] and ML-style expressions.

- Our type system allows open code templates irrespective of surrounding environments. Code template

  ```
  ‘(x+1)
  ```

  has type $\Box(\{x : \mathit{int}\}\rho \triangleright \mathit{int})$, meaning that the code template can be executed or composed in any environment that has at least program variable `x` of integer type. Environment $\{x : \mathit{int}\}\rho$ denotes a type environment that may have entries other than $\{x : \mathit{int}\}$. The postfix ρ is a *row variables* in record typing [21] that ranges over a finite set of field types.
- Our type system supports unrestricted imperative operations for open code templates. Our type system accepts the following program:

  ```
  let val a = ref ‘1
      val f = ‘(fn x -> ,(a := ‘(x + 1); ‘2))
   in ! a end
  ```

  where program variable `a` has type $\Box(\{x : \mathit{int}\}\rho \triangleright \mathit{int})$ `ref`.

  This example is from Calcagno, Moggi and Sheard [2] where it is untypable because their type system restricts imperative operations only to closed code.
- Our type system supports unhygienic manipulation of code, where the programmers intentionally let code's free variables be captured at runtime. Unhygienic code templates are frequently used for both programming brevity and performance.

  Our type system, for example, accepts the following `map` function that generates a specialized code for the list argument:

  ```
  fun smap [] = ‘[]
    | smap (x::r) = ‘((f ,x) :: ,(smap r))
  fun map ls = eval ‘(fn f => ,(smap ls))
  ```

  Note that the `smap` function generates open code whose free variable `f` will be bound after `smap`'s recursive calls. When using closed code only, we need as many extra closures and applications as the length of `ls`.
- Our type system supports hygienic code manipulation [14] too that avoids the capture of free variables of open code templates during code composition. Suppose we define the `or` function as follows. The definition does not work as expected because of the variable-capture.

  ```
  fun or a b =
     ‘(let val v = ,a in if v then v else ,b end)
  ```

  When invoking `(or ‘false ‘`v`)`, we obtain an unintended code

  ```
  ‘(let val v = false in if v then v else v end).
  ```

  Since the free variable v in the arguments is captured by v declared in the `or` function, the residual code returns `false`, not argument v.

  To avoid such unintended variable-capture during code composition we use a new language construct to generate fresh variable names: hygienic abstraction $\lambda^* v.e$ that substitutes a fresh name for v inside e before its application. Our type system accepts the following correct version of the `or` function:

  ```
  fun or a b =‘((λ*v.if v then v else ,b) ,a)
  ```

  Similarly, the `h2` function in Taha's thesis [24] can be defined and type-checked in our target language by using the hygienic abstraction λ^*:

  ```
  fun h2 n z =
    if n=0 then z
    else ‘((λ*x. ,(h2 (n-1) ‘(x + ,z))) ,(lift n))
  ```

  Note that call `(h2 2 ‘4)` evaluates to

  ```
  ‘((λx1.(λx2.x2+(x1+4)) 1) 2)
  ```

  not to `‘((λx.(λx.x+(x+4)) 1) 2)`.

  Note that `h2`'s unhygienic version is one where λx is used instead of $\lambda^* x$. The programmer can choose between the two versions of `h2`, depending on his/her intention, and our type system accepts both.
- Our type system uses record subtypes for the types of free variables in open code templates. For example,

```
if e then `(x+1) else `1
```

has type $\Box(\{x : \mathit{int}\}\rho \triangleright \mathit{int})$ because the else-branch's type $\Box(\rho' \triangleright \mathit{int})$ is a record subtype of the then-branch's type $\Box(\{x : int\}\rho \triangleright \mathit{int})$ (the type operator $\triangleright$ is contravariant, like $\rightarrow$, over its left-hand side record type).

- Our type system orthogonally combines the let-polymorphism with open code templates. Consider

```
let val x = `y in `(,x + 1); `((,x 1) + z) end
```

Our type system assigns to `x` a polymorphic open code type $\forall\alpha\forall\rho.\Box(\{y : \alpha\}\rho\triangleright\alpha)$, which is distinctly instantiated for each occurrence of `x`. The first occurrence of `x` has $\Box(\{y : \mathit{int}\} \triangleright \mathit{int})$ and the second has $\Box(\{y : \mathit{int} \rightarrow \mathit{int}, z : \mathit{int}\} \triangleright \mathit{int} \rightarrow \mathit{int})$.

Organization Section 2 introduces notation. Section 3 presents a language λ^{sim}_{open} and its simple type system. Section 4 presents a polymorphic extension λ^{poly}_{open} of λ^{sim}_{open} and its polymorphic type system. Section 5 presents a principal type inference algorithm for λ^{poly}_{open}. Section 6 shows the relation of our type system to other existing ones. Section 7 concludes. Representative parts of the proofs for key lemmas of this article appear in [13].

2. Notation

$A \oplus B$ is the union of disjoint sets A and B whereas $A \cup B$ is the union of arbitrary sets A and B. $[i..j]$ is the set of integers from i to j, and is empty if $i > j$. $\{x_i\}_{i=j}^{k}$ is $\{x_j, x_{j+1}, \ldots, x_k\}$, and is empty if $j > k$. $\{x_i\}_{i=j}^{k}$ is written $\{x_i\}_j^k$ without confusion.

A stage number n is a non-negative integer. We assume that any expression for a stage number is always non-negative.

A relation R from set A to set B is any subset of the Cartesian product of A and B, i.e. $R \subseteq A \times B$. A relation $F \subseteq A \times B$ is a (partial) function if for all $a \in A$, $\{b \mid a : b \in F\}$ contains at most one element. The set of functions from set A to set B is denoted by $A \rightarrow B$. The set of functions from finite subsets of A to set B is denoted by $A \xrightarrow{fin} B$. The empty function is denoted by $\varnothing$. For a function F, $\mathit{dom}(F) = \{a \mid a : b \in F\}$ and $\mathit{range}(F) = \{b \mid a : b \in F\}$. $F|_A$ and $F|_A^-$ are the restrictions of a function F such that

$$F|_A = \{a : b \in F \mid a \in A\} \text{ and } F|_A^- = \{a : b \in F \mid a \notin A\}.$$

$F_1 :: F_2$ is the union of domain-disjoint functions F_1 and F_2:

$$F_1 :: F_2 = \{a : b \mid a : b \in F_1 \text{ or } a : b \in F_2\},$$

where $\mathit{dom}(F_1) \cap \mathit{dom}(F_2) = \varnothing$. $F + a : b$ is the extension of a function F with $a : b$ such that $F + a : b = F|_{\{a\}}^- :: \{a : b\}$. $F_1 \cdots F_m$ is the sequence of functions $F_1, \ldots, F_m$. $F_1 \cdots F_m + a : b$ is the sequence of functions $F_1, \cdots, F_{m-1}, F_m + a : b$.

3. Simply-Typed Multi-Staged Language λ^{sim}_{open}

This section presents the multi-staged language λ^{sim}_{open} and its simple type system. λ^{sim}_{open} supports open code templates, unrestricted operations on references, both hygienic and unhygienic code manipulation, and lifting values into code.

3.1 Syntax

$$\begin{array}{rcl} e \in \mathit{Exp} & ::= & c \mid x \mid \lambda x.e \mid e_1 e_2 \\ & & \mid \texttt{box}\, e \mid \texttt{unbox}_k\, e \mid \texttt{open}\, e \mid \texttt{lift}\, e \mid \lambda^* x.e \\ & & \mid \texttt{ref}\, e \mid \texttt{!}\, e \mid e_1 \texttt{:=} e_2 \\ & & \mid \texttt{clos}(x, e, \mathcal{E}) \mid l \end{array}$$

The syntax of λ^{sim}_{open} is based on the implicit $\lambda^{\Box}$ language [7, 8], and has additional expressions to manipulate code templates and to support imperative operations. Expression c is a constant of base type. Expressions $\texttt{box}\, e$ and $\texttt{unbox}_k\, e$ are for manipulating code templates, and respectively correspond to the backquote (`), comma (,) ($k > 0$) and the `eval` ($k = 0$) in Lisp's quasi-quote notation. Expression $\texttt{open}\, e$ is useful for type inference by guiding the places of subtyping, and has no effect on evaluation. Expression $\texttt{lift}\, e$ is for converting the value of e into its corresponding code template. For hygienic code manipulation [14], expression $\lambda^* x.e$ renames x (of the same stage level) before a code is plugged inside e at runtime. Unless inside code templates, $\lambda^* x.e$ is not different from $\lambda x.e$. Expressions $\texttt{ref}\, e$, $\texttt{!}\, e$ and $e_1 \texttt{:=} e_2$ are conventional for imperative operations. Expression $\texttt{clos}(x, e, \mathcal{E})$ is the closure of $\lambda x.e$ in an environment $\mathcal{E}$. Location l is a store location. Closures and locations, which are values, are included as expressions for convenience for the type soundness proof.

3.2 Operational Semantics

Multi-staged language λ^{sim}_{open} has a call-by-value semantics. Figure 1 shows the big-step operational semantics. For brevity we do not present the error-generating rules, which are standard. Evaluation

$$\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}')$$

means that expression e under environment $\mathcal{E}$, store $\mathcal{S}$ and variables $\mathcal{V}$ evaluates to result r, store $\mathcal{S}'$ and variables $\mathcal{V}'$ at stage n.

Values are expressions which cannot be reduced further. In multi-staged languages, values exist at every stage. Values at each stage are called staged values. A staged value v^n ($n > 0$) is a frozen expression that is to be evaluated later when it is demoted to stage 0 by the $\texttt{unbox}_0$ construct.

$$\begin{array}{lll} \text{Staged Values} & v^n \in V^n & \\ & ::= c \mid \texttt{box}\, v^1 \mid \texttt{clos}(x, e, \mathcal{E}) \mid l & \text{if } n = 0 \\ & ::= c \mid x \mid \lambda x.v^n \mid v^n v^n & \\ & \quad \mid \texttt{box}\, v^{n+1} \mid \texttt{unbox}_k\, v^{n-k} & \\ & \quad \mid \texttt{open}\, v^n \mid \texttt{lift}\, v^n \mid \lambda^* x.v^n & \\ & \quad \mid \texttt{ref}\, v^n \mid \texttt{!}\, v^n \mid v_1^n \texttt{:=} v_2^n & \\ & \quad \mid \texttt{clos}(x, e, \mathcal{E}) \mid l & \text{if } n > k \geq 0 \end{array}$$

Variables	x, y, z, w	$\in$	*Var*	$= \mathcal{V}_E \oplus \mathcal{V}_I$
Locations	l	$\in$	*Loc*	$=$ *a set of locations*
Stores	$\mathcal{S}$	$\in$	*Store*	$= \mathit{Loc} \xrightarrow{fin} V^0$
Environments	$\mathcal{E}$	$\in$	*Env*	$= \mathit{Var} \xrightarrow{fin} V^0$
Closures	$\texttt{clos}(x, e, \mathcal{E})$	$\in$	*Clos*	$= \mathit{Var} \times \mathit{Exp} \times \mathit{Env}$
Results	r	$\in$	*Res*	$= \bigcup_{n=0}^{\infty} V^n \oplus \{\texttt{err}\}$

Var is a countable set of variable names and is the disjoint union of $\mathcal{V}_E$ and $\mathcal{V}_I$. $\mathcal{V}_E$ is a set of external variable names that may appear in the source program, while $\mathcal{V}_I$ is a set of internal variable names that $\lambda^* x.e$ may generate at runtime. The disjointness of $\mathcal{V}_I$ and $\mathcal{V}_E$ ensures that the internal variable names generated by $\lambda^* x.e$ are always different from the external variable names in the source program. $\mathcal{V}_I$ is simply denoted by $\mathcal{V}$ in the big-step operational semantics of λ^{sim}_{open}. We use x, y, z, w for variable names, and w for an internal variable name alone. A store $\mathcal{S}$ is a finite function from locations to values of stage 0. An environment $\mathcal{E}$ is a finite function from variables to values of stage 0. Expression $\texttt{clos}(x, e, \mathcal{E})$ is the closure of $\lambda x.e$ in an environment $\mathcal{E}$. The result r is either a value or `err`.

LEMMA 3.1. *$V^n \subset V^{n+1}$ for $n \geq 0$.*

PROOF By induction on a staged value v^n in V^n. □

LEMMA 3.2. *(Result) If $\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}')$ then r is in $V^n \oplus \{\texttt{err}\}$.*

PROOF By induction on the derivation of $\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}')$. Lemma 3.1 is needed in cases of $\texttt{unbox}_k\, e$ and $\texttt{lift}\, e$ ($n = 0$). □

(ECON) $$\mathcal{E},\mathcal{S},\mathcal{V} \vdash c \xrightarrow{n} (c,\mathcal{S},\mathcal{V})$$

(EVAR) $$\frac{x \in dom(\mathcal{E})}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash x \xrightarrow{0} (\mathcal{E}(x),\mathcal{S},\mathcal{V})} \qquad \mathcal{E},\mathcal{S},\mathcal{V} \vdash x \xrightarrow{n+1} (x,\mathcal{S},\mathcal{V})$$

(EABS) $$\mathcal{E},\mathcal{S},\mathcal{V} \vdash \lambda x.e \xrightarrow{0} (\texttt{clos}(x,e,\mathcal{E}),\mathcal{S},\mathcal{V})$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{n+1} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \lambda x.e \xrightarrow{n+1} (\lambda x.v,\mathcal{S}',\mathcal{V}')}$$

(EAPP) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e_1 \xrightarrow{0} (\texttt{clos}(x,e,\mathcal{E}'),\mathcal{S}_1,\mathcal{V}_1) \quad \mathcal{E},\mathcal{S}_1,\mathcal{V}_1 \vdash e_2 \xrightarrow{0} (v_2,\mathcal{S}_2,\mathcal{V}_2) \quad \mathcal{E}'+x:v_2,\mathcal{S}_2,\mathcal{V}_2 \vdash e \xrightarrow{0} (v_3,\mathcal{S}_3,\mathcal{V}_3)}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e_1 e_2 \xrightarrow{0} (v_3,\mathcal{S}_3,\mathcal{V}_3)}$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e_1 \xrightarrow{n+1} (v_1,\mathcal{S}_1,\mathcal{V}_1) \quad \mathcal{E},\mathcal{S}_1,\mathcal{V}_1 \vdash e_2 \xrightarrow{n+1} (v_2,\mathcal{S}_2,\mathcal{V}_2)}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e_1 e_2 \xrightarrow{n+1} (v_1 v_2,\mathcal{S}_2,\mathcal{V}_2)}$$

(EBOX) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{n+1} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{box}\ e \xrightarrow{n} (\texttt{box}\ v,\mathcal{S}',\mathcal{V}')}$$

(EEVAL) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{0} (\texttt{box}\ v^1,\mathcal{S}_1,\mathcal{V}_1) \quad \mathcal{E},\mathcal{S}_1,\mathcal{V}_1 \vdash v^1 \xrightarrow{0} (v^0,\mathcal{S}_2,\mathcal{V}_2)}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{unbox}_0\ e \xrightarrow{0} (v^0,\mathcal{S}_2,\mathcal{V}_2)}$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{n+1} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{unbox}_0\ e \xrightarrow{n+1} (\texttt{unbox}_0\ v,\mathcal{S}',\mathcal{V}')}$$

(EUNBOX) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{0} (\texttt{box}\ v,\mathcal{S}',\mathcal{V}') \quad k>0}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{unbox}_k\ e \xrightarrow{k} (v,\mathcal{S}',\mathcal{V}')}$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{n+1} (v,\mathcal{S}',\mathcal{V}') \quad k>0}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{unbox}_k\ e \xrightarrow{n+1+k} (\texttt{unbox}_k\ v,\mathcal{S}',\mathcal{V}')}$$

(EOPEN) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{0} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{open}\ e \xrightarrow{0} (v,\mathcal{S}',\mathcal{V}')}$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{n+1} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{open}\ e \xrightarrow{n+1} (\texttt{open}\ v,\mathcal{S}',\mathcal{V}')}$$

(ELIFT) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{0} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{lift}\ e \xrightarrow{0} (\texttt{box}\ v,\mathcal{S}',\mathcal{V}')}$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{n+1} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{lift}\ e \xrightarrow{n+1} (\texttt{lift}\ v,\mathcal{S}',\mathcal{V}')}$$

(EGENSYM) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \lambda w.([x^n \overset{n}{\mapsto} w]\ e) \xrightarrow{n} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \oplus \{w\} \vdash \lambda^* x.e \xrightarrow{n} (v,\mathcal{S}',\mathcal{V}')}$$

(EREF) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{0} (v,\mathcal{S}',\mathcal{V}') \quad l \notin dom(\mathcal{S}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{ref}\ e \xrightarrow{0} (l,\mathcal{S}'+l:v,\mathcal{V}')}$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{n+1} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{ref}\ e \xrightarrow{n+1} (\texttt{ref}\ v,\mathcal{S}',\mathcal{V}')}$$

(EDEREF) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{0} (l,\mathcal{S}',\mathcal{V}') \quad l \in dom(\mathcal{S}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{!}\ e \xrightarrow{0} (\mathcal{S}'(l),\mathcal{S}',\mathcal{V}')}$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e \xrightarrow{n+1} (v,\mathcal{S}',\mathcal{V}')}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{!}\ e \xrightarrow{n+1} (\texttt{!}\ v,\mathcal{S}',\mathcal{V}')}$$

(EASSIGN) $$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e_1 \xrightarrow{0} (l,\mathcal{S}_1,\mathcal{V}_1) \quad \mathcal{E},\mathcal{S}_1,\mathcal{V}_1 \vdash e_2 \xrightarrow{0} (v,\mathcal{S}_2,\mathcal{V}_2)}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e_1 \texttt{:=} e_2 \xrightarrow{0} (v,\mathcal{S}_2+l:v,\mathcal{V}_2)}$$

$$\frac{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e_1 \xrightarrow{n+1} (v_1,\mathcal{S}_1,\mathcal{V}_1) \quad \mathcal{E},\mathcal{S}_1,\mathcal{V}_1 \vdash e_2 \xrightarrow{n+1} (v_2,\mathcal{S}_2,\mathcal{V}_2)}{\mathcal{E},\mathcal{S},\mathcal{V} \vdash e_1 \texttt{:=} e_2 \xrightarrow{n+1} (v_1 \texttt{:=} v_2,\mathcal{S}_2,\mathcal{V}_2)}$$

(ELOC) $$\mathcal{E},\mathcal{S},\mathcal{V} \vdash l \xrightarrow{n} (l,\mathcal{S},\mathcal{V})$$

(ECLOS) $$\mathcal{E},\mathcal{S},\mathcal{V} \vdash \texttt{clos}(x,e,\mathcal{E}') \xrightarrow{n} (\texttt{clos}(x,e,\mathcal{E}'),\mathcal{S},\mathcal{V})$$

Figure 1. Big-step operational semantics of λ^{sim}_{open}: $\mathcal{V}$ means $\mathcal{V}_I$.

$$\begin{array}{lll}
[x^n \overset{m}{\mapsto} w]\ c & = c & \\
[x^n \overset{m}{\mapsto} w]\ y & = w, & \text{if } x=y \text{ and } m=n \\
 & = y, & \text{otherwise} \\
[x^n \overset{m}{\mapsto} w]\ \lambda y.e & = \lambda y.e, & \text{if } x=y \text{ and } m=n \\
 & = \lambda y.([x^n \overset{m}{\mapsto} w]\ e), & \text{otherwise} \\
[x^n \overset{m}{\mapsto} w]\ (e_1 e_2) & = ([x^n \overset{m}{\mapsto} w]\ e_1)\ ([x^n \overset{m}{\mapsto} w]\ e_2) & \\
[x^n \overset{m}{\mapsto} w]\ \texttt{box}\ e & = \texttt{box}\ ([x^n \overset{m+1}{\mapsto} w]\ e) & \\
[x^n \overset{m}{\mapsto} w]\ \texttt{unbox}_k\ e & = \texttt{unbox}_k\ ([x^n \overset{m-k}{\mapsto} w]\ e) & \\
[x^n \overset{m}{\mapsto} w]\ \texttt{open}\ e & = \texttt{open}\ ([x^n \overset{m}{\mapsto} w]\ e) & \\
[x^n \overset{m}{\mapsto} w]\ \texttt{lift}\ e & = \texttt{lift}\ ([x^n \overset{m}{\mapsto} w]\ e) & \\
[x^n \overset{m}{\mapsto} w]\ \lambda^* y, e & = \lambda^* y.e, & \text{if } x=y \text{ and } m=n \\
 & = \lambda^* y.([x^n \overset{m}{\mapsto} w]\ e), & \text{otherwise} \\
[x^n \overset{m}{\mapsto} w]\ \texttt{ref}\ e & = \texttt{ref}\ ([x^n \overset{m}{\mapsto} w]\ e) & \\
[x^n \overset{m}{\mapsto} w]\ (\texttt{!}\ e) & = \texttt{!}\ ([x^n \overset{m}{\mapsto} w]\ e) & \\
[x^n \overset{m}{\mapsto} w]\ (e_1 \texttt{:=} e_2) & = ([x^n \overset{m}{\mapsto} w]\ e_1) \texttt{:=} ([x^n \overset{m}{\mapsto} w]\ e_2) & \\
[x^n \overset{m}{\mapsto} w]\ \texttt{clos}(y,e,\mathcal{E}) & = \texttt{clos}(y,e,\mathcal{E}), & \text{if } x=y \text{ and } m=n \\
 & = \texttt{clos}(y,[x^n \overset{m}{\to} w]\ e,\mathcal{E}), & \text{otherwise} \\
[x^n \overset{m}{\mapsto} w]\ l & = l &
\end{array}$$

Figure 2. Substituting w for free variable x of stage n at stage m

λ^{sim}_{open} extends the traditional lambda calculus conservatively. At stage 0, (ECON), (EVAR), (EABS) and (EAPP) are exactly the same as the call-by-value semantics of lambda calculus. Alpha-conversion and beta-reduction are available whenever necessary at stage 0. (EREF), (EDEREF) and (EASSIGN) are for usual imperative operations at stage 0.

λ^{sim}_{open} can construct, compose and evaluate code templates at runtime. (EBOX) constructs a code template. λ^{sim}_{open} allows open expressions inside a code template. The stage number increases by one as we go inside a code template. If the stage number is positive, we are inside a code template. (EUNBOX) merges code templates at stage $n-k$ into the current code at stage n. (EUNBOX) provides a way to escape from a code template temporarily before the code template ends. At stage 0, (EEVAL) converts a code template $\texttt{box}v^1$ into expression v^1 and then evaluates v^1, which is restricted to closed code by the type system (See Subsection 3.3). No variable in a code template is bound at stage 0 since v^1 is always closed in (EEVAL).

By default λ^{sim}_{open} does not perform alpha-conversion inside code templates. To capture free variables at code composition, we need to preserve the names of free variables inside code templates. Note that the meaning of an expression may change after alpha-conversion inside code templates. However, there is no restriction on alpha-conversion at stage 0; as only closed code is evaluated in (EEVAL), code templates cannot bind variables of stage 0. For example, $\lambda x.\texttt{unbox}_1\ y$ and $\lambda z.\texttt{unbox}_1\ y$ are alpha congruent at stage 1, but their meanings are different:

$$\{y:\texttt{box}\ x\},\mathcal{S},\mathcal{V} \vdash \lambda x.\texttt{unbox}_1\ y \xrightarrow{1} (\lambda x.x,\mathcal{S},\mathcal{V})$$

whereas

$$\{y:\texttt{box}\ x\},\mathcal{S},\mathcal{V} \vdash \lambda z.\texttt{unbox}_1\ y \xrightarrow{1} (\lambda z.x,\mathcal{S},\mathcal{V}).$$

On the other hand, $\lambda x.\texttt{unbox}_0 y$ and $\lambda z.\texttt{unbox}_0 y$ are alpha congruent at stage 0, and they have the same meaning since y is closed code.

λ^{sim}_{open} provides support for explicit alpha-conversion inside code templates for hygienic manipulation [14]. (EGENSYM) substitutes a fresh variable w for variable x in $\lambda^* x.e$ at stage n, and then evaluates the renamed lambda expression. At stage 0, $\lambda^* x.e$ has no effect on evaluation since it is just alpha-conversion in normal lambda calculus. For example, the following two expressions evaluate to

different results:

$$\{y : \texttt{box}\ x\}, \mathcal{S}, \mathcal{V} \vdash \lambda x.\texttt{unbox}_1\ y \xrightarrow{1} (\lambda x.x, \mathcal{S}, \mathcal{V})$$

whereas

$$\{y : \texttt{box}\ x\}, \mathcal{S}, \mathcal{V} \vdash \lambda^* x.\texttt{unbox}_1\ y \xrightarrow{1} (\lambda w.x, \mathcal{S}, \mathcal{V})$$

for some fresh internal variable w. Figure 2 shows the definition of the staged renaming operator $[x^n \overset{m}{\mapsto} w]$. In `(EGENSYM)`, only the occurrences of x at stage n are replaced with w, because binders only act at the current stage. For example, in $\lambda x.\texttt{box}\ x$ at stage n, the binder acts at stage n but the occurrence of x in $\texttt{box}x$ is at stage $n+1$.

λ^{sim}_{open} has additional features to manipulate code templates. `(ELIFT)` lifts values to corresponding code templates. `(ELIFT)` is the reverse of `(EEVAL)`. Note that a staged value v^n can also be v^{n+1} (Lemma 3.1). `(EOPEN)` has no effect on evaluation. The `open` construct is a syntactic marker to which our type inference algorithm applies subtyping (See Section 3.3). Inside a code template, `(EABS)` reduces the body code in a lambda abstraction. Lambda expressions are not values but expressions inside a code template.

3.3 Type System

The key idea of the type system of λ^{sim}_{open} is to include a type environment in a code type. Type environments inside code types enable us to type open code templates, avoiding the restriction of Calcagno et al.'s systems [3, 2] that the types of free variables (even inside code values) are always looked up in the current type environment. Our type system supports imperative operations for open code templates, and both hygienic and unhygienic code manipulation.

Types	A, B	$\in$	*Type*		
Type Environments	Γ	$\in$	*TyEnv*	$=$	$\mathit{Var} \overset{fin}{\to} \mathit{Type}$
Store Typings	Σ	$\in$	*ST*	$=$	$\mathit{Loc} \overset{fin}{\to} \mathit{Type}$

Figure 3 shows a type system for λ^{sim}_{open} in the style of Harper [11]; all type rules need store typing in order to support imperative operations. We use A, B for types. A type environment Γ is a finite function from variables to types. A store typing Σ is a finite function from locations to types.

$$\text{Types} \qquad A, B \quad ::= \quad \iota \mid A \to B \mid \Box(\Gamma \triangleright A) \mid A\ \texttt{ref}$$

We use ι for base type, $A \to B$ for function types, and $\Box(\Gamma \triangleright A)$ for code template types. $\Box(\Gamma \triangleright A)$ is a conditional modal type in which the condition Γ specifies the types of free program variables in the code template of type A. We use $A\ \texttt{ref}$ for types of store locations having A-typed values.

DEFINITION 3.1. *A store $\mathcal{S}$ is well typed with respect to a store typing Σ, written $\models \mathcal{S} : \Sigma$, if and only if $dom(\mathcal{S}) = dom(\Sigma)$ and $\Sigma; \varnothing \vdash \mathcal{S}(l) : \Sigma(l)$ for every $l \in dom(\mathcal{S})$.*

DEFINITION 3.2. *An environment $\mathcal{E}$ is well typed with respect to a store typing Σ and a type environment Γ, written $\Sigma \models \mathcal{E} : \Gamma$, if and only if $dom(\mathcal{E}) = dom(\Gamma)$ and $\Sigma; \varnothing \vdash \mathcal{E}(x) : \Gamma(x)$ for every $x \in dom(\mathcal{E})$.*

The typing judgment

$$\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : A$$

means that an expression e, under store typing Σ and type environments $\Gamma_0 \cdots \Gamma_n$, has type A at stage n. $\Gamma_0 \cdots \Gamma_n$ is a sequence of type environments $\Gamma_0, \ldots, \Gamma_n$. Γ_n is the current type environment. Subscripts $0, \ldots, n$ are stage numbers.

`(TSCON)` $$\Sigma; \Gamma_0 \cdots \Gamma_n \vdash c : \iota$$

`(TSVAR)` $$\frac{\Gamma_n(x) = A}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash x : A}$$

`(TSABS)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n + x : A \vdash e : B}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash \lambda x.e : A \to B}$$

`(TSAPP)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e_1 : A \to B \quad \Sigma; \Gamma_0 \cdots \Gamma_n \vdash e_2 : A}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e_1 e_2 : B}$$

`(TSBOX)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\ e : \Box(\Gamma \triangleright A)}$$

`(TSUNBOX)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : \Box(\Gamma_{n+k} \triangleright A) \quad k > 0}{\Sigma; \Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash \texttt{unbox}_k e : A}$$

`(TSEVAL)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : \Box(\varnothing \triangleright A)}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash \texttt{unbox}_0\ e : A}$$

`(TSLIFT)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : A}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash \texttt{lift}\ e : \Box(\Gamma \triangleright A)}$$

`(TSGENSYM)` $$\frac{\begin{array}{l}\Sigma; \Gamma_0 \cdots \Gamma_n + w : A \vdash [x^n \overset{n}{\mapsto} w]\ e : B \\ \quad w \text{ is a fresh prog. var. in } (\Sigma, \Gamma_0 \cdots \Gamma_n, \lambda^* x.e)\end{array}}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash \lambda^* x.e : A \to B}$$

`(TSOPEN)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : \Box(\varnothing \triangleright A)}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash \texttt{open}\ e : \Box(\Gamma \triangleright A)}$$

`(TSREF)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : A}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash \texttt{ref}\ e : A\ \texttt{ref}}$$

`(TSDEREF)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : A\ \texttt{ref}}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash\ !\ e : A}$$

`(TSASSIGN)` $$\frac{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e_1 : A\ \texttt{ref} \quad \Sigma; \Gamma_0 \cdots \Gamma_n \vdash e_2 : A}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e_1 \texttt{:=} e_2 : A}$$

`(TSLOC)` $$\frac{\Sigma(l) = A}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash l : A\ \texttt{ref}}$$

`(TSCLOS)` $$\frac{\Sigma \models \mathcal{E} : \Gamma \quad \Sigma; \Gamma + x : A \vdash e : B}{\Sigma; \Gamma_0 \cdots \Gamma_n \vdash \texttt{clos}(x, e, \mathcal{E}) : A \to B}$$

Figure 3. Type system for λ^{sim}_{open}

`(TSBOX)` makes open code template `box` e typable since

$$\Sigma; \Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A$$

may hold for an open expression e. Type $\Box(\Gamma \triangleright A)$ is for code template `box` e, indicating that e has type A in all accessible stages satisfying Γ. Note that the code template type has a subtype property: if $\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\ e : \Box(\Gamma \triangleright A)$ holds, then $\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\ e : \Box(\Gamma' \triangleright A)$ also holds for $\Gamma' \supseteq \Gamma$. `(TSUNBOX)` checks if a code template from the previous stage is properly captured by the type environment at the current stage. `(TSUNBOX)` shows that code template types have the modal property: if e has type $\Box(\Gamma \triangleright A)$ at stage Γ_n, then $\texttt{unbox}_k\ e$ has type A in an accessible stage satisfying Γ. `(TSEVAL)` allows only closed code to be evaluated by the `eval` construct. Type $\Box(\varnothing \triangleright A)$ is the same as type $\Box A$ in Davies and Pfenning [7, 8]. `(TSOPEN)` relaxes closed code types to behave as open ones. Expression `open` e, which is first used in [3], induces a syntax-driven subtyping, by weakening the number of free variables. Without `open`, closed code cannot be used both in evaluation and composition if the composition context requires a non-empty type environment. For instance, consider

$$\begin{array}{l}\textit{let } u = \texttt{box}\ 1 \textit{ in} \\ \textit{let } v = \texttt{box}\ (\lambda x.(\texttt{unbox}_1(\texttt{open}\ u)) + x) \textit{ in} \\ (\texttt{unbox}_0\ u) + 1\end{array}$$

While u's type is $\square(\varnothing \triangleright \mathit{int})$ because u is used in evaluation "$(\texttt{unbox}_0\, u)+1$", the "$\texttt{open}\, u$" relaxes u's type to $\square(\{x : \mathit{int}\} \triangleright \mathit{int})$ so as to allow u to be composed into a place whose context has free x. Such relaxation not only expands the set of typable expressions but also enables a syntax-driven principal type inference algorithm (See Section 5).

(TSGENSYM) says that the type of $\lambda^* x.e$ is the same as the type of the expression resulting from renaming its bound variable x by a fresh program variable w. (TSGENSYM) requires the explicit alpha-conversion hence the type of $\lambda^* x.e$ can be different from that of $\lambda x.e$. For example,

$$\varnothing; \{y : \square(\{x : \mathit{int}\} \triangleright \mathit{int})\}\ \ \{x : \mathit{int}\} \vdash \lambda x.\texttt{unbox}_1\ y : \mathit{int} \to \mathit{int},$$

$$\varnothing; \{y : \square(\{x : \mathit{int}\} \triangleright \mathit{int})\}\ \ \{x : \mathit{int}\} \vdash \lambda^* x.\texttt{unbox}_1\ y : A \to \mathit{int}$$

for some A. Hence,

$$\varnothing; \{y : \square(\{x : \mathit{int}\} \triangleright \mathit{int})\}\ \ \{x : \mathit{int}\} \vdash (\lambda x.\texttt{unbox}_1\ y)\ \mathit{true}$$

is untypable, but

$$\varnothing; \{y : \square(\{x : \mathit{int}\} \triangleright \mathit{int})\}\ \ \{x : \mathit{int}\} \vdash (\lambda^* x.\texttt{unbox}_1\ y)\ \mathit{true} : \mathit{int}.$$

(TSLIFT) allows typing the code template that corresponds to the value of e. Lemma 3.1 shows that a value of stage n is also a value of stage $n+1$. Hence, if e has type A, then e's value is also an A-typed value at a higher stage under any context, hence $\square(\Gamma \triangleright A)$ for an arbitrary Γ.

Unlike the type system of Calcagno et al. [2], (TSREF), (TSDEREF) and (TSASSIGN) support imperative operations for open code templates without any restriction. (TSCON), (TSVAR), (TSABS) and (TSAPP) are as usual except for extending to multi-staged setting.

3.4 Soundness of the Type System

First, consider (EEVAL) at stage 0. (EEVAL) converts $\texttt{box}\ v^1$ into v^1 at stage 0. By the following demotion lemma, if

$$\Sigma; \varnothing \vdash \texttt{box}\ v^1 : \square(\Gamma \triangleright A)$$

then $\Sigma; \Gamma \vdash v^1 : A$ because $\Sigma; \varnothing\Gamma \vdash v^1 : A$ by (TSBOX). Hence, code template of type $\square(\Gamma \triangleright A)$ can be safely converted into expressions of type A under type environment Γ.

LEMMA 3.3. *(Demotion) If* $\Sigma; \varnothing\Gamma_1 \cdots \Gamma_n \vdash v : A$ *for* $n > 0$*, then* $\Sigma; \Gamma_1 \cdots \Gamma_n \vdash v : A$.

PROOF By induction on the type derivation of $\Sigma; \varnothing\Gamma_1 \cdots \Gamma_n \vdash v : A$. □

Second, a well typed expression preserves its type after evaluation in λ^{sim}_{open}. For the proof of the preservation lemma, we need the result lemma (Lemma 3.2) and the demotion lemma (Lemma 3.3).

LEMMA 3.4. *(Preservation) If* $\models \mathcal{S} : \Sigma,\ \Sigma \models \mathcal{E} : \Gamma_0$,

$$\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : A \ \textit{and}\ \mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}'),$$

then $\Sigma'; \varnothing\Gamma_1 \cdots \Gamma_n \vdash r : A$ *and* $\models \mathcal{S}' : \Sigma'$ *for some* $\Sigma' \supseteq \Sigma$.

PROOF By induction on the type derivation $\Sigma; \Gamma_0 \cdots \Gamma_n \vdash e : A$ and evaluation $\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}')$. □

Finally, the type system of λ^{sim}_{open} is sound. If a closed expression is well typed, then it preserves the type after evaluation. Hence, the evaluation result can not be err because err is not typable.

THEOREM 3.1. *(Soundness) If*

$$\varnothing; \varnothing \vdash e : A \ \textit{and}\ \varnothing, \varnothing, \mathcal{V} \vdash e \xrightarrow{0} (r, \mathcal{S}, \mathcal{V}'),$$

then $\Sigma; \varnothing \vdash r : A$ *where* $\models \mathcal{S} : \Sigma$.

PROOF Immediate from the preservation lemma. □

$$
\begin{array}{lll}
[x^n \overset{m}{\mapsto} w]\ \texttt{let}\ (y\ e_1)\ e_2 & & \\
\quad = \quad \texttt{let}\ (y\ ([x^n \overset{m}{\mapsto} w]\ e_1))\ e_2, & & \text{if } x = y \text{ and } m = n \\
\quad = \quad \texttt{let}\ (y\ ([x^n \overset{m}{\mapsto} w]\ e_1))\ [x^n \overset{m}{\mapsto} w]\ e_2, & & \text{otherwise} \\
\textit{Other cases} & \textit{The same as } \lambda^{\mathrm{sim}}_{\mathrm{open}} \textit{ in Figure 2} &
\end{array}
$$

Figure 4. Substitution at stage m of w for free variable x declared at stage n

$$
\begin{array}{ll}
\texttt{(ELET)} & \dfrac{\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e_1 \xrightarrow{0} (v_1, \mathcal{S}_1, \mathcal{V}_1) \quad \mathcal{E} + x : v_1, \mathcal{S}_1, \mathcal{V}_1 \vdash e_2 \xrightarrow{0} (v_2, \mathcal{S}_2, \mathcal{V}_2)}{\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash \texttt{let}\ (x\ e_1)\ e_2 \xrightarrow{0} (v_2, \mathcal{S}_2, \mathcal{V}_2)} \\
 & \\
 & \dfrac{\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e_1 \xrightarrow{n+1} (v_1, \mathcal{S}_1, \mathcal{V}_1) \quad \mathcal{E}, \mathcal{S}_1, \mathcal{V}_1 \vdash e_2 \xrightarrow{n+1} (v_2, \mathcal{S}_2, \mathcal{V}_2)}{\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash \texttt{let}\ (x\ e_1)\ e_2 \xrightarrow{n+1} (\texttt{let}\ (x\ v_1)\ v_2, \mathcal{S}_2, \mathcal{V}_2)} \\
\textit{Other rules} & \textit{The same as } \lambda^{\mathrm{sim}}_{\mathrm{open}} \textit{ in Figure 1}
\end{array}
$$

Figure 5. Big-step operational semantics of λ^{poly}_{open}: $\mathcal{V}$ is a set of internal program variables $\mathcal{V}_I$.

4. Polymorphic Multi-Staged Language λ^{poly}_{open}

This section presents a let-polymorphic multi-staged language λ^{poly}_{open} and its polymorphic type system. The type system adopts the let-polymorphism where type generalization occurs only in let expressions. This polymorphic generalization is applied to the record types that are used for the type environments accompanying code types. The record types are generalized in a different axis too, in the sense of record subtyping in order to relax the constraints imposed by the type environment. Syntactic value restriction is used to support imperative operations in the polymorphic type system.

4.1 Syntax

λ^{poly}_{open} extends λ^{sim}_{open} by the let-binding expression $\texttt{let}\ (x\ e_1)\ e_2$ that binds the value of e_1 to x in e_2.

$$
\begin{array}{rcl}
e \in \mathit{Exp} & ::= & c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \texttt{let}\ (x\ e_1)\ e_2 \\
 & & \mid \texttt{box}\ e \mid \texttt{unbox}_k\ e \mid \texttt{open}\ e \mid \texttt{lift}\ e \mid \lambda^* x.e \\
 & & \mid \texttt{ref}\ e \mid\ !\ e \mid e_1 := e_2 \\
 & & \mid \texttt{clos}(x, e, \mathcal{E}) \mid l
\end{array}
$$

4.2 Operational Semantics

Figure 5 shows the big-step operational semantics of $\texttt{let}(x\ e_1)\ e_2$. The other evaluation rules are the same as those for λ^{sim}_{open}. The error generation rules are omitted.

$$
\begin{array}{lll}
\text{Staged Values } v^n \in V^n & & \\
\quad ::= \ c \mid \texttt{box}\ v^1 \mid \texttt{clos}(x, e, \mathcal{E}) \mid l & & \text{if } n = 0 \\
\quad ::= \ c \mid x \mid \lambda x.v^n \mid v^n v^n & & \\
\quad \mid \texttt{box}\ v^{n+1} \mid \texttt{unbox}_k\ v^{n-k} & & \\
\quad \mid \texttt{open}\ v^n \mid \texttt{lift}\ v^n \mid \lambda^* x.v^n & & \\
\quad \mid \texttt{ref}\ v^n \mid\ !\ v^n \mid v_1^n := v_2^n & & \\
\quad \mid \texttt{let}\ (x\ v^n)\ v^n & & \\
\quad \mid \texttt{clos}(x, e, \mathcal{E}) \mid l & & \text{if } n > k \geq 0
\end{array}
$$

λ^{poly}_{open} has $\texttt{let}\ (x\ v^n)\ v^n$ as additional staged values. Other staged values are the same as in λ^{sim}_{open}. Except for staged values, the other semantic domains *Loc*, *Store*, *Var*, *Env*, *Clos* and *Res* are the same as in λ^{sim}_{open} (See Section 3.2).

LEMMA 4.1. $V^n \subset V^{n+1}$ *for* $n \geq 0$.

PROOF By induction on a staged value v^n in V^n. □

$$
\begin{array}{lcll}
\mathrm{expansive}^n(c) & = & \mathit{False} & \\
\mathrm{expansive}^n(x) & = & \mathit{False} & \\
\mathrm{expansive}^n(\lambda x.e) & = & \mathit{False}, & \text{if } n = 0 \vee e \in V^1 \\
 & = & \mathit{True}, & \text{otherwise} \\
\mathrm{expansive}^n(e_1 e_2) & = & \mathit{True} & \\
\mathrm{expansive}^n(\mathtt{box}\ e) & = & \mathit{False}, & \text{if } e \in V^1 \\
 & = & \mathit{True}, & \text{otherwise} \\
\mathrm{expansive}^n(\mathtt{unbox}_k\ e) & = & \mathit{True}, & n \geq k \\
\mathrm{expansive}^n(\mathtt{open}\ e) & = & \mathrm{expansive}^n(e) & \\
\mathrm{expansive}^n(\mathtt{lift}\ e) & = & \mathrm{expansive}^n(e) & \\
\mathrm{expansive}^n(\lambda^* x.e) & = & \mathit{False}, & \text{if } n = 0 \vee e \in V^1 \\
 & = & \mathit{True}, & \text{otherwise} \\
\mathrm{expansive}^n(\mathtt{ref}\ e) & = & \mathit{True} & \\
\mathrm{expansive}^n(\mathtt{!}\ e) & = & \mathrm{expansive}^n(e) & \\
\mathrm{expansive}^n(e_1 \mathtt{:=} e_2) & = & \mathrm{expansive}^n(e_1) \vee \mathrm{expansive}^n(e_2) & \\
\mathrm{expansive}^n(\mathtt{let}\ (x\ e_1)\ e_2) & = & \mathrm{expansive}^n(e_1) \vee \mathrm{expansive}^n(e_2) & \\
\mathrm{expansive}^n(\mathtt{clos}(x, e, \mathcal{E})) & = & \mathit{False} & \\
\mathrm{expansive}^n(l) & = & \mathit{False} &
\end{array}
$$

Figure 6. Staged predicate $\mathrm{expansive}^n(e)$: a non-expansive expression at stage n is syntactically guaranteed never to expand the store during its evaluation at stage m for every $m \leq n$.

LEMMA 4.2. *(Result) If* $\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}')$ *then* r *is in* $V^n \oplus \{\mathtt{err}\}$.

PROOF By induction on the derivation of $\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}')$. □

4.3 Staged, Syntactic Value Restriction

Let-polymorphic generalization is activated only when expression e_1 in $\mathtt{let}\ (x\ e_1)\ e_2$ never expands the store. We need to statically check whether an expression expands the store or not.

One thing we have to be careful in devising such a check is that a non-expansive expression at stage n should not expand the store during its evaluation at any stage $m \leq n$. This *demotion-closedness* is because any expression at stage n can be demoted by the $\mathtt{unbox}_0$ construct to stage m for some $m < n$ until it evaluates to a value of stage 0.

Staged predicate $\mathrm{expansive}^n(e)$ in Figure 6 is designed based on Wright [27]: if an expression is a syntactic value, then it never expands the store and hence it is a non-expansive expression. Note that $\mathrm{expansive}^n(e)$ satisfies the *demotion-closedness*. Consider the cases of $\lambda x.e$ and $\lambda^* x.e$. If e is in V^1, then $\lambda x.e$ and $\lambda^* x.e$ are in V^1. Then, from Lemma 4.1, $\lambda x.e$ and $\lambda^* x.e$ are in V^n for any $n > 0$ (and hence they never expand the store at stage n). Meanwhile, at stage 0, $\lambda x.e$ and $\lambda^* x.e$ never expand the store; they just reduce to closures. Thus, if e is in V^1 or at stage 0, $\lambda x.e$ and $\lambda^* x.e$ never expand the store for any stage n. The $\mathtt{box}\ e$ case is similar as follows. If $e \in V^1$, then, $\mathtt{box} e \in V^0$ by definition of V^n. By Lemma 4.1, $\mathtt{box}\ e \in V^0 \subset V^{n+1}$ for $n \geq 0$. Hence, $\mathtt{box}\ e$ is a value at any stage; it is a non-expansive expression. $\mathtt{ref}\ e$ expands the store at stage 0. Hence, $\mathrm{expansive}^n(\mathtt{ref}\ e)$ is defined to be true for any stage in order to satisfy *demotion-closedness*. $e_1 e_2$ may expand the store at stage 0. Hence, $\mathrm{expansive}^n(e_1 e_2)$ is also defined to be true for any stage. Expression $\mathtt{unbox}_k\ e$ may expand the store at stage k. Hence, $\mathrm{expansive}^n(\mathtt{unbox}_k e)$ is true for any stage $n \geq k$. At stage $n < k$, $\mathtt{unbox}_k\ e$ is meaningless since stage number is always non-negative. Expressions $\mathtt{!}\, e_1$, $e_1 \mathtt{:=} e_2$, $\mathtt{open} e_1$, $\mathtt{lift}\ e_1$ and $\mathtt{let}\ (x\ e_1)\ e_2$ expand the store if e_1 or e_2 expands the store. Expressions c, x $\mathtt{clos}(x, e, \mathcal{E})$ and l do not expand the store for any stage. Note that $\mathrm{expansive}^n(\mathtt{clos}(x, e, \mathcal{E}))$ and $\mathrm{expansive}^n(l)$ are not used in type inference because they are values not occurring in the source program. They are included just for our proofs.

4.4 Type System

Our polymorphic type system extends λ^{sim}_{open} by ML's let-polymorphism and Rémy's record type [21].

$$
\begin{array}{lcll}
\text{Types} & A, B & \in & \mathit{Type} \\
 & & ::= & \iota \mid A \to B \mid \Box(\Gamma \triangleright A) \mid A\ \mathtt{ref} \mid \alpha \\
\text{Type Variables} & \alpha, \beta & \in & \mathit{TyVar} \\
\text{Fields} & F, G & \in & \mathit{Field} \quad = \quad \mathit{Type} \oplus \{\bot\} \\
\text{Field Variables} & \theta & \in & \mathit{FieldVar} \\
\text{Store Typings} & \Sigma & \in & \mathit{StoTy} \quad = \quad \mathit{Loc} \xrightarrow{fin} \mathit{Type} \\
\text{Type Environments} & \Gamma & \in & \mathit{TyEnv} \quad = \quad \mathit{Var} \xrightarrow{fin} \mathit{Field} \\
 & & ::= & \{x_i : F_i\}_1^m \mid \{x_i : F_i\}_1^k \rho_L \\
 & & & \text{where } L = \{x_i\}_1^k \\
\text{Type Environments Variables} & \rho & \in & \mathit{TyEnvVar}
\end{array}
$$

We use A, B for types, and α, β for type variables. *Field*[21] is either *Type* or $\bot$. We use F, G for fields, and θ for field variables. Store typing Σ is a finite function from locations to types. Type environment Γ is a finite function from variables to fields, and can be regarded as a record. We use ρ for type environment variables. Type environment $\{x_i : F_i\}_1^m \rho_L$ denotes a type environment

$$\{x_i : F_i\}_1^m :: \{y_i : G_i\}_1^k \text{ for some } \{y_i : G_i\}_1^k$$

and $::$ is a domain-disjoint union. Operator $::$ extends for type environment variables as

$$\{x_i : F_i\}_1^m :: \rho = \{x_i : F_i\}_1^m \rho.$$

In a type environment, a field is a type or $\bot$. If $x : \bot$ in Γ, x is an undefined variable in Γ. (Such extension to $\bot$, following the idea in Rémy's record typing [21], allows the record field addition that is necessary only for the type inference algorithm in Section 5.) Hence,

$$\{x_i : F_i\}_1^k \text{ and } \{x_i : F_i\}_1^k :: \{y : \bot\}$$

are regarded as the same type environment. Similarly, for a field variable θ,

$$\rho_L \text{ and } \{y : \theta\} :: \rho_{L \oplus \{y\}}$$

are regarded as the same type environment. Without confusion, we write $\{x_i : F_i\}_1^k \rho$ instead of $\{x_i : F_i\}_1^k \rho_L$.

$$
\begin{array}{lcl}
\xi, \chi & \in & \mathit{TyVar} \oplus \mathit{TyEnvVar} \oplus \mathit{FieldVar} \\
X, Y & \in & \mathit{Type} \oplus \mathit{TyEnv} \oplus \mathit{Field}
\end{array}
$$

Some symbols range over multiple sets: ξ, χ are used for type variables, type environment variables, or field variables. X, Y are used for types, type environments, or fields.

$$
\begin{array}{lcll}
\text{Type Schemes} & \tau, \sigma & \in \mathit{TyScheme} & ::= \quad \forall \xi.\tau \mid A \\
\text{Field Schemes} & \mu, \pi & \in \mathit{FieldScheme} & = \quad \mathit{TypeScheme} \oplus \{\bot\} \\
\text{Type Scheme Env} & \Delta & \in \mathit{TySchemeEnv} & = \quad \mathit{Var} \xrightarrow{fin} \mathit{FieldScheme} \\
 & & & ::= \quad \{x_i : \mu_i\}_1^m \mid \{x_i : \mu_i\}_1^k \rho_L \\
 & & & \text{where } L = \{x_i\}_1^k
\end{array}
$$

We use τ, σ for type schemes, and μ, π for field schemes. Type schemes are in the prenex form, containing outermost quantification only. $\forall \xi.\tau$ binds ξ in τ. A type variable α, field variable θ or a type environment variable ρ is free in τ if it occurs in τ and is not bound. A type scheme $\forall \xi_1 \ldots \forall \xi_m.\sigma$ is written $\forall \xi_1 \ldots \xi_m.\sigma$. Type scheme environment is a finite function from variables to field schemes. Type scheme environment is an extension of type environment using field schemes. Similarly to type environments, type environment schemes may contain type environment variables. Type scheme environment $\{x_i : \mu_i\}_1^m \rho_L$ denotes a type scheme environment

$$\{x_i : \mu_i\}_1^m :: \{y_i : \pi_i\}_1^k \text{ for some } \{y_i : \pi_i\}_1^k.$$

Note that a variable ρ is not for type scheme environments but for type environments, which are monomorphic. Like type environments,

$$\{x_i : \mu_i\}_1^k \text{ and } \{x_i : \mu_i\}_1^k :: \{y : \bot\}$$

are the same, and for a field variable θ,

$$\rho_L \text{ and } \{y : \theta\} :: \rho_{L \oplus \{y\}}$$

are the same.

DEFINITION 4.1. *(Free variables)* $\mathrm{FV}(\tau)$ *is the set of free type variables, free field variables and free type environment variables occurring in* τ. FV *is abused for type schemes, field schemes, store typings and type scheme environments:*

$$\mathrm{FV}(\Sigma) \;=\; \bigcup_{A \in \mathit{range}(\Sigma)} \mathrm{FV}(A)$$

and

$$\begin{array}{rll} \mathrm{FV}(\Delta) & = \bigcup_{i=1}^m \mathrm{FV}(\mu_i) \cup \{\rho\}, & \text{if } \Delta = \{x_i : \mu_i\}_1^m \rho \\ & = \bigcup_{i=1}^m \mathrm{FV}(\mu_i), & \text{if } \Delta = \{x_i : \mu_i\}_1^m. \end{array}$$

DEFINITION 4.2. *(Bound variables)* $\mathrm{BV}(\tau)$ *is the set of bound type variables, bound field variables and bound type environment variables occurring in* τ. BV *is abused for type schemes, field schemes and type scheme environments:*

$$\mathrm{BV}(\Delta) = \textstyle\bigcup_{i=1}^m \mathrm{BV}(\mu_i)$$

where $\Delta = \{x_i : \mu_i\}_1^m \rho$ *or* $\Delta = \{x_i : \mu_i\}_1^m$.

Substitutions $R, S, T, U \quad \in \quad \mathit{TySub}$
$= (\mathit{FieldVar} \xrightarrow{\mathit{fin}} \mathit{Field}) :: (\mathit{TyVar} \xrightarrow{\mathit{fin}} \mathit{Type}) :: (\mathit{TyEnvVar} \xrightarrow{\mathit{fin}} \mathit{TyEnv})$

A substitution is a finite function from field variables to fields, from type variables to types, and from type environment variables to type environments. Applying substitution R to X is written RX. The composition of substitutions S followed by R is written $R \cdot S$ or RS, and is defined as

$$\{\xi : R(S(\xi)) \mid \xi \in \mathit{dom}(S)\} :: \{\xi : R(\xi) \mid \xi \in \mathit{dom}(R) \setminus \mathit{dom}(S)\}.$$

DEFINITION 4.3. *(Type instantiation) A type scheme* $\forall \xi_1 \ldots \xi_m.A$ *instantiates to a type* B*, written* $\forall \xi_1 \ldots \xi_m.A \succ B$ *if and only if there exists a substitution* S *with domain* $\{\xi_i\}_1^m$ *and* $SA = B$.

DEFINITION 4.4. *(Type environment instantiation) A type scheme environment* $\{x_i : \mu_i\}_1^m \rho$ *instantiates to a type environment* Γ*, written* $\{x_i : \mu_i\}_1^m \rho \succ \Gamma$*, if and only if* $\Gamma = \{x_i : F_i\}_1^m \rho$ *and* $\mu_i \succ F_i$ *for* $i \in [1..m]$. *Likewise,* $\{x_i : \mu_i\}_1^m$ *instantiates to* Γ *if and only if* $\Gamma = \{x_i : F_i\}_1^m$ *and* $\mu_i \succ F_i$ *for* $i \in [1..m]$.

DEFINITION 4.5. *A store* $\mathcal{S}$ *is well typed with respect to a store typing* Σ*, written* $\models \mathcal{S} : \Sigma$*, if and only if* $\mathit{dom}(\mathcal{S}) = \mathit{dom}(\Sigma)$ *and* $\Sigma; \varnothing \vdash \mathcal{S}(l) : \Sigma(l)$ *for every* $l \in \mathit{dom}(\mathcal{S})$.

DEFINITION 4.6. *An environment* $\mathcal{E}$ *is well typed with respect to a store typing* Σ *and a type environment scheme* Δ*, written* $\Sigma \models \mathcal{E} : \Delta$*, if and only if*

$$\mathit{dom}(\mathcal{E}) = \mathit{dom}(\Delta) \;\; \textit{and} \;\; \Sigma; \varnothing \vdash \mathcal{E}(x) : A$$

for every $x \in \mathit{dom}(\Delta)$ *and every* A *where* $\Delta(x) \succ A$.

Typing judgment

$$\Sigma; \Delta_0 \cdots \Delta_n \vdash e : A$$

means that an expression e, under store typing Σ and type scheme environments $\Delta_0 \cdots \Delta_n$, has type A at stage n. $\Delta_0 \cdots \Delta_n$ is a sequence of type scheme environments $\Delta_0, \ldots, \Delta_n$. Δ_n is the current type scheme environment. Subscripts $0, \ldots, n$ are stage numbers.

(TCON) $$\Sigma; \Delta_0 \cdots \Delta_n \vdash c : \iota$$

(TVAR) $$\frac{\Delta_n(x) \succ A}{\Sigma; \Delta_0 \cdots \Delta_n \vdash x : A}$$

(TABS) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n + x : A \vdash e : B}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \lambda x.e : A \to B}$$

(TAPP) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \vdash e_1 : A \to B \quad \Sigma; \Delta_0 \cdots \Delta_n \vdash e_2 : A}{\Sigma; \Delta_0 \cdots \Delta_n \vdash e_1 e_2 : B}$$

(TBOX) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \Gamma \vdash e : A}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{box}\; e : \Box(\Gamma \triangleright A)}$$

(TUNBOX) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \vdash e : \Box(\Gamma \triangleright A) \quad \Delta_{n+k} \succ \Gamma \quad k > 0}{\Sigma; \Delta_0 \cdots \Delta_n \cdots \Delta_{n+k} \vdash \mathtt{unbox}_k\; e : A}$$

(TEVAL) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \vdash e : \Box(\varnothing \triangleright A)}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{unbox}_0\; e : A}$$

(TOPEN) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \vdash e : \Box(\varnothing \triangleright A)}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{open}\; e : \Box(\Gamma \triangleright A)}$$

(TLIFT) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \vdash e : A}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{lift}\; e : \Box(\Gamma \triangleright A)}$$

(TGENSYM) $$\frac{\begin{array}{c}\Sigma; \Delta_0 \cdots \Delta_n + w : A \vdash [x^n \overset{n}{\mapsto} w]\, e : B \\ w \text{ not in } (\Sigma, \Delta_0 \cdots \Delta_n, \lambda^* x.e')\end{array}}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \lambda^* x.e : A \to B}$$

(TREF) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \vdash e : A}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{ref}\; e : A\; \mathtt{ref}}$$

(TDEREF) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \vdash e : A\; \mathtt{ref}}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{!}\; e : A}$$

(TASSIGN) $$\frac{\Sigma; \Delta_0 \cdots \Delta_n \vdash e_1 : A\; \mathtt{ref} \quad \Sigma; \Delta_0 \cdots \Delta_n \vdash e_2 : A}{\Sigma; \Delta_0 \cdots \Delta_n \vdash e_1 \mathtt{:=} e_2 : A}$$

(TLOC) $$\frac{\Sigma(l) = A}{\Sigma; \Delta_0 \cdots \Delta_n \vdash l : A\; \mathtt{ref}}$$

(TCLOS) $$\frac{\Sigma \models \mathcal{E} : \Delta \quad \Sigma; \Delta + x : A \vdash e : B}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{clos}(x, e, \mathcal{E}) : A \to B}$$

(TLETIMP) $$\frac{\begin{array}{c}\mathrm{expansive}^n(e_1) \\ \Sigma; \Delta_0 \cdots \Delta_n \vdash e_1 : A \quad \Sigma; \Delta_0 \cdots \Delta_n + x : A \vdash e_2 : B\end{array}}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{let}\; (x\; e_1)\; e_2 : B}$$

(TLETAPP) $$\frac{\begin{array}{l}\neg\, \mathrm{expansive}^n(e_1) \\ \Sigma; \Delta_0 \cdots \Delta_n \vdash e_1 : A \\ \Sigma; \Delta_0 \cdots \Delta_n + x : \mathit{GEN}_A(\Sigma, \Delta_0 \cdots \Delta_n) \vdash e_2 : B\end{array}}{\Sigma; \Delta_0 \cdots \Delta_n \vdash \mathtt{let}\; (x\; e_1)\; e_2 : B}$$

$\mathit{GEN}_A(\Sigma, \Delta_0 \cdots \Delta_n) = \forall \xi_1 \ldots \xi_m.A$ such that
$\{\xi_1, \ldots, \xi_m\} = \mathrm{FV}(A) \setminus (\mathrm{FV}(\Sigma) \cup \bigcup_{i=0}^n \mathrm{FV}(\Delta_i))$

Figure 7. Polymorphic type system for $\lambda^{\mathit{poly}}_{\mathit{open}}$

(TBOX), (TOPEN) and (TLIFT) restrict code template type $\Box(\Gamma \triangleright A)$ to be conditioned by monomorphic type environment Γ. This restriction, which is analogous to rank-1 polymorphism, allows us to avoid the impossible *principal typing* in the Hindley/Milner-style type inference [26].

(TBOX), (TOPEN) and (TLIFT) are the rules that may introduce type environment variables. Consider the (TBOX) case. Suppose that

$$\Sigma; \Delta_0 \cdots \Delta_n \{x_i : A_i\}_1^m \vdash e : A.$$

Then,

$$\Sigma; \Delta_0 \cdots \Delta_n \{x_i : A_i\}_1^m :: \{y_i : F_i\}_1^k \vdash e : A \text{ for some } \{y_i : F_i\}_1^k.$$

Hence, using a type environment variable ρ,

$$\Sigma; \Delta_0 \cdots \Delta_n \{x_i : A_i\}_1^m \rho \vdash e : A$$

or

$$\Sigma; \Delta_0 \cdots \Delta_n \vdash \texttt{box}\, e : \Box(\{x_i : A_i\}_1^m \rho \triangleright A).$$

(TOPEN) and (TLIFT) also introduce type environment variables as follows.

$$\Sigma; \Delta_0 \cdots \Delta_n \vdash \texttt{open}\, e : \Box(\rho \triangleright A) \text{ for some } \rho \text{ and}$$
$$\Sigma; \Delta_0 \cdots \Delta_n \vdash \texttt{lift}\, e : \Box(\rho \triangleright A) \text{ for some } \rho.$$

In typing judgment $\Sigma; \Delta_0 \cdots \Delta_n \vdash e : A$, Δ_0 always has the form of $\{x_i : \tau_i\}_1^m$ (not $\{x_i : \tau_i\}_1^m \rho$) while Δ_i $(i > 0)$ may be $\{y_i : \mu_i\}_1^k \rho$. This is because type environment variables are introduced only into code template types (by (TBOX), (TOPEN) and (TLIFT)) and the type environment $\{x_i : F_i\}_1^m \rho$ in a code template type $\Box(\{x_i : F_i\}_1^m \rho \triangleright A)$ can be Δ_i $(i > 0)$ in a type derivation.

(TLETAPP) allows the let-polymorphism in $\texttt{let}\ (x\ e_1)\ e_2$ if e_1 guarantees not to expand the store (if $\text{expansive}^n(e_1)$ is false). Note that type environment variables and field variables, as well as type variables, can be generalized in (TLETAPP). If the evaluation of e_1 does not expand the store, it is safe to generalize free type variables, free type environment variables, or free field variables in the type of e_1. In (TVAR) and (TUNBOX), such generalized type variables, type environment variables and field variables respectively instantiate to some types, type environments and fields. For example, the type environment variable ρ in $\forall\rho.\Box(\{x_i : F_i\}_1^m \rho \triangleright A)$ can instantiate to some type environment $\{y_i : G_i\}_1^k$ (or $\{y_i : G_i\}_1^k \rho'$) where $\{x_i\}_1^m \cap \{y_i\}_1^k = \varnothing$. In (TLETIMP), e_1 may expand the store, the type of e_1 is not generalized. Other rules are the same as in λ_{open}^{sim} except for this polymorphic extension.

4.5 Soundness of the Type System

(EEVAL) at stage 0 converts $\texttt{box}\, v^1$ into v^1 at stage 0; it demotes values at stage n to expressions at stage $(n-1)$. The following demotion lemma shows that demotion preserves types.

LEMMA 4.3. *(Demotion) If* $\Sigma; \varnothing\Delta_1 \cdots \Delta_n \vdash v : A$ *where* $n > 0$*, then*

$$\Sigma; \Delta_1 \cdots \Delta_n \vdash v : A.$$

PROOF By induction on the type derivation of $\Sigma; \varnothing\Delta_1 \cdots \Delta_n \vdash v : A$. Predicate $\text{expansive}^n(e)$'s demotion-closedness is needed in the proof. □

An expression preserves its type after evaluation in λ_{open}^{poly}. For the proof of the preservation lemma, we need the result lemma (Lemma 4.2) and the demotion lemma (Lemma 4.3).

LEMMA 4.4. *(Preservation) If* $\models \mathcal{S} : \Sigma$, $\Sigma \models \mathcal{E} : \Delta_0$, *If*

$$\Sigma; \Delta_0 \cdots \Delta_n \vdash e : A, \text{ and } \mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}'),$$

then,

$$\Sigma'; \varnothing\Delta_1 \cdots \Delta_n \vdash r : A \text{ and } \models \mathcal{S}' : \Sigma' \text{ for some } \Sigma' \supseteq \Sigma.$$

PROOF By induction on the type derivation of $\Sigma; \Delta_0 \cdots \Delta_n \vdash e : A$ and evaluation $\mathcal{E}, \mathcal{S}, \mathcal{V} \vdash e \xrightarrow{n} (r, \mathcal{S}', \mathcal{V}')$. □

Finally, the type system of λ_{open}^{poly} is sound. If a closed expression is well typed, then it preserves the type after evaluation. Hence, the evaluation result can not be err. Note that err is not typable.

THEOREM 4.1. *(Soundness) If* $\varnothing; \varnothing \vdash e : A$ *and* $\varnothing, \varnothing, \mathcal{V} \vdash e \xrightarrow{0} (r, \mathcal{S}, \mathcal{V}')$, *then* $\Sigma; \varnothing \vdash r : A$ *and* $\models \mathcal{S} : \Sigma$.

PROOF Immediate from the preservation lemma. □

5. Type Inference Algorithm

This section presents a sound and complete type inference algorithm that finds the principal types for λ_{open}^{poly}. Figure 10 shows the type inference algorithm for λ_{open}^{poly}. Algorithm $\texttt{infer}(\Delta_0 \cdots \Delta_n, e, A, \mathcal{Q})$ takes as input a sequence of type scheme environments $\Delta_0 \cdots \Delta_n$, an expression e, a type A and a set $\mathcal{Q}$ of fresh type, type environment or field variables to be used in the infer algorithm. The infer algorithm finds the most general substitution R satisfying

$$\varnothing; (R\Delta_0) \cdots (R\Delta_n) \vdash e : RA.$$

Figure 11 shows the extension of type environment or field addition operation in record types. In $\varnothing; \Delta_0 \cdots \Delta_n \vdash e : A$, Δ_i is a record type which may contain ρ variable for $i > 0$. In (IABS), (IGENSYM) and (ILET), we need to extend Δ_n with $x : \tau$, meaning that $\Delta_n + x : \tau$ has $x : \tau$ whether Δ_n already has some $x : \theta$ or not. To support such field addition in record types, we adopt Rémy's record types [21] where a field variable can be $\bot$. Meanwhile, Rémy uses the different notion for the field in record types such as $x : \texttt{pre}(A)$ (not $x : A$) or $x : \text{abs}$ (not $x : \bot$).

$\texttt{unify}(E, \mathcal{Q}) =$
  We use Rémy's unification algorithm [21, 20].
  The only difference is:
    For equation $\Box(\Gamma_1 \triangleright A_1) = \Box(\Gamma_2 \triangleright A_2)$ during unification,
      we convert it into $\Gamma_1 = \Gamma_2$ and $A_1 = A_2$.
    For equation $A_1\ \texttt{ref} = A_2\ \texttt{ref}$ during unification,
      we convert it into $A_1 = A_2$.

Figure 8. Unification algorithm

Our unification algorithm (Figure 8) is basically the same as Rémy's [21, 20] except for the code and reference types. Remy's unification algorithm takes as input a set of equations for types, fields or records (type environments). The second input $\mathcal{Q}$ is the set of fresh variables that will be used during the unification. The algorithm returns the most general unifier [21, 20].

(IBOX), (IOPEN) and (ILIFT) introduce type environment variables into code template types. As mentioned in Section 4, in a sequence of type scheme environments $\Delta_0 \cdots \Delta_n$, Δ_n $(n > 0)$ may be $\{x_i : F_i\}_1^m \rho$. Hence, to infer the type of x, (IVAR) checks $\Delta_n \succ \{x : A\}\rho$ (not $\Delta_n(x) \succ A$) because it may be that $\Delta_n = \{x_i : F_i\}_1^m \rho'$ and $x \notin \{x_i\}_1^m$. (ILET) generalizes free type environment variables and free field variables, as well as free type variables. Such generalized type, field or type environment variables instantiate to concrete types, fields or type environments in (IVAR) and (IUNBOX). (IVAR) and (IUNBOX) need the instantiation of type scheme environments, whose algorithm is presented in Figure 9. (IBOX) and (IUNBOX) relate two different staged type scheme environments with each other by making the sequence of type scheme environments longer or shorter. (IGENSYM) executes the explicit name change before type inference. (ICON), (IABS), (IAPP), (IREF), (IDEREF) and (IASSIGN) are straightforward except for multi-staged setting.

The infer algorithm is a sound and complete type inference algorithm for λ_{open}^{poly}. The completeness lemma (Lemma 5.2) establishes that the infer algorithm finds the principal types of λ_{open}^{poly}.

LEMMA 5.1. *(Soundness) Suppose that*

$$\mathcal{Q} \cap (\text{FV}(A) \cup \bigcup_{i=0}^{n} \text{FV}(\Delta_i)) = \varnothing.$$

If $\texttt{infer}(\Delta_0 \cdots \Delta_n, e, A, \mathcal{Q})$ *succeeds with S, then*

$$\varnothing; (S\Delta_0) \cdots (S\Delta_n) \vdash e : SA.$$

PROOF By induction on each case of the infer algorithm. □

(* $\texttt{inst}(\Delta, \mathcal{Q})$ finds a type environment Γ that satisfies $\Delta \succ \Gamma$
using fresh variables in $\mathcal{Q}$. *)

$\texttt{inst}(\Delta, \mathcal{Q}) =$
make distinct all bound variables in Δ by renaming the bound variables
if $\Delta = \{x_i : \mu_i\}_1^m \rho$ then
let S be a substitution such that $\mathit{dom}(S) = \bigoplus_{i=1}^m \mathrm{BV}(\mu_i)$ and
$\mathit{range}(S) \subseteq \mathcal{Q}$ in
let $\mu_i \succ F_i$ by S for $i \in [1..m]$ in
$\{x_i : F_i\}_1^m \rho$
if $\Delta = \{x_i : \mu_i\}_1^m$ then
let S be a substitution such that $\mathit{dom}(S) = \bigoplus_{i=1}^m \mathrm{BV}(\mu_i)$ and
$\mathit{range}(S) \subseteq \mathcal{Q}$ in
let $\mu_i \succ F_i$ by S for $i \in [1..m]$ in
$\{x_i : F_i\}_1^m$

Figure 9. Instantiation of type scheme environments

LEMMA 5.2. *(Completeness) Suppose that*

$$\varnothing; (R\Delta_0) \cdots (R\Delta_n) \vdash e : RA$$

and $\mathcal{Q} \cap (\mathrm{FV}(A) \cup \bigcup_{i=0}^n \mathrm{FV}(\Delta_i)) = \varnothing$.
Then, $\texttt{infer}(\Delta_0 \cdots \Delta_n, e, A, \mathcal{Q})$ *succeeds with* S *such that* $R|_{\mathcal{Q}}^- = TS|_{\mathcal{Q}}^-$ *for some* T *and* $\mathrm{RV}(S) \subseteq \mathcal{Q} \cup \mathrm{FV}(A) \cup \bigcup_{i=0}^n \mathrm{FV}(\Delta_i)$.

PROOF By induction on the derivation of $\varnothing; (R\Delta_0) \cdots (R\Delta_n) \vdash e : RA$. The equality $R|_{\mathcal{Q}}^- = TS|_{\mathcal{Q}}^-$ is modulo the aforementioned equivalence between type environments with the tailing ρ variables: for a field variable θ, ρ_L and $\{y : \theta\} :: \rho_{L \oplus \{y\}}$ are equivalent. □

Suppose that a type variable α is not in a set of type and type environment variables $\mathcal{Q}$. Then $\texttt{infer}(\varnothing, e, \alpha, \mathcal{Q})$ always terminates for any expression e. If $\texttt{infer}(\varnothing, e, \alpha, \mathcal{Q})$ succeeds with S, then $\varnothing; \varnothing \vdash e : S\alpha$. Moreover, if there is a substitution R such that $\varnothing; \varnothing \vdash e : R\alpha$, then $R|_{\mathcal{Q}}^- = TS|_{\mathcal{Q}}^-$ for some T. Hence, $S\alpha$ is the principal type of e.

6. Relation to Other Multi-Staged Languages

This section compares λ^{poly}_{open} (or λ^{sim}_{open}) with other multi-staged languages such as the implicit $\lambda^{\square}$ [7, 8], λ^{i}_{let} [3], $\lambda^{\bigcirc}$ [6] and λ^{+}_{code} [4].

6.1 Relation to the Implicit $\lambda^{\square}$

Typed expressions in the implicit $\lambda^{\square}$ [7, 8] can be embedded into λ^{sim}_{open}. Figure 12 shows the type system of the implicit $\lambda^{\square}$. The translation is straightforward because the implicit $\lambda^{\square}$ accepts only closed code templates while λ^{sim}_{open} accepts code templates containing free variables. Expression translation from the implicit $\lambda^{\square}$ to λ^{sim}_{open} is:

$$\begin{array}{cc} [\![x]\!] = x & [\![\lambda x.e]\!] = \lambda x.[\![e]\!] \\ [\![e_1 e_2]\!] = [\![e_1]\!][\![e_2]\!] & [\![\texttt{box}\ e]\!] = \texttt{box}\ [\![e]\!] \\ [\![\texttt{unbox}_0\ e]\!] = \texttt{unbox}_0 [\![e]\!] & [\![\texttt{unbox}_k\ e]\!] = \texttt{unbox}_k\ (\texttt{open}\ [\![e]\!]) \\ & \text{where } k > 0. \end{array}$$

Type translation from the implicit $\lambda^{\square}$ to λ^{sim}_{open} is:

$$[\![\iota]\!] = \iota \quad [\![A \to B]\!] = [\![A]\!] \to [\![B]\!] \quad [\![\square A]\!] = \square(\varnothing \triangleright [\![A]\!]).$$

Type environment translation $[\![\Gamma]\!]$ is point-wise:

$$[\![\{x_1 : A_1, \ldots, x_n : A_n\}]\!] = \{x_1 : [\![A_1]\!], \ldots, x_n : [\![A_n]\!]\}.$$

LEMMA 6.1. *If* $\Gamma_0 \cdots \Gamma_n \vdash^i e : A$ *in implicit* $\lambda^{\square}$, *then*

$$\varnothing; [\![\Gamma_0]\!] \cdots [\![\Gamma_n]\!] \vdash [\![e]\!] : [\![A]\!]$$

in $\lambda^{\mathrm{sim}}_{\mathrm{open}}$.

$\texttt{infer}(\Delta_0 \cdots \Delta_n, c, A, \mathcal{Q}) =$ (ICON)
$\texttt{unify}(\{A = \iota\}, \mathcal{Q})$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, x, A, \{\rho\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2) =$ (IVAR)
let $\Gamma_n = \texttt{inst}(\Delta_n, \mathcal{Q}_1)$ in
$\texttt{unify}(\{\Gamma_n = \{x : A\}\rho\}, \mathcal{Q}_2)$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \lambda x.e, A, \{\alpha, \beta\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2 \oplus \mathcal{Q}_3) =$ (IABS)
let $(S_1, \Delta_1' \cdots \Delta_n') = \texttt{add}(\Delta_0 \cdots \Delta_n, \{x : \alpha\}, \mathcal{Q}_1)$ in
let $S_2 = \texttt{infer}(\Delta_0' \cdots \Delta_n', e, S_1\beta, \mathcal{Q}_2)$ in
let $S_3 = \texttt{unify}(\{S_2 S_1 A = S_2 S_1 \alpha \to S_2 S_1 \beta\}, \mathcal{Q}_3)$ in $S_3 S_2 S_1$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, e_1 e_2, A, \{\alpha\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2) =$ (IAPP)
let $S_1 = \texttt{infer}(\Delta_0 \cdots \Delta_n, e_1, \alpha \to A, \mathcal{Q}_1)$ in
let $S_2 = \texttt{infer}((S_1\Delta_0) \cdots (S_1\Delta_n), e_2, S_1\alpha, \mathcal{Q}_2)$ in $S_2 S_1$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \texttt{box}\ e, A, \{\rho, \alpha\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2) =$ (IBOX)
let $S_1 = \texttt{infer}(\Delta_0 \cdots \Delta_n \rho, e, \alpha, \mathcal{Q}_1)$ in
let $S_2 = \texttt{unify}(\{S_1 A = \square(S_1\rho \triangleright S_1\alpha)\}, \mathcal{Q}_2)$ in $S_2 S_1$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \texttt{unbox}_k\ e, A, \{\rho\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2 \oplus \mathcal{Q}_3) =$ (IUNBOX)
let $\Gamma_n = \texttt{inst}(\Delta_n, \mathcal{Q}_1)$ in
let $S_2 = \texttt{infer}(\Delta_0 \cdots \Delta_{n-k}, e, \square(\rho \triangleright A), \mathcal{Q}_2)$ in
let $S_3 = \texttt{unify}(\{S_2\Gamma_n = S_2\rho\}, \mathcal{Q}_3)$ in $S_3 S_2$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \texttt{unbox}_0\ e, A, \mathcal{Q}) =$ (IEVAL)
$\texttt{infer}(\Delta_0 \cdots \Delta_n, e, \square(\varnothing \triangleright A), \mathcal{Q})$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \texttt{open}\ e, A, \{\rho, \alpha\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2) =$ (IOPEN)
let $S_1 = \texttt{infer}(\Delta_0 \cdots \Delta_n, e, \square(\varnothing \triangleright \alpha), \mathcal{Q}_1)$ in
let $S_2 = \texttt{unify}(\{S_1 A = \square(\rho \triangleright S_1\alpha)\}, \mathcal{Q}_2)$ in $S_2 S_1$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \texttt{lift}\ e, A, \{\rho, \alpha\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2) =$ (ILIFT)
let $S_1 = \texttt{infer}(\Delta_0 \cdots \Delta_n, e, \alpha, \mathcal{Q}_1)$ in
let $S_2 = \texttt{unify}(\{S_1 A = \square(\rho \triangleright S_1\alpha)\}, \mathcal{Q}_2)$ in $S_2 S_1$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \lambda^* x.e, A, \{\alpha, \beta\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2 \oplus \mathcal{Q}_3) =$ (IGENSYM)
w is a fresh internal program variable
let $(S_1, \Delta_0' \cdots \Delta_n') = \texttt{add}(\Delta_0 \cdots \Delta_n, \{w : \alpha\}, \mathcal{Q}_1)$ in
let $S_2 = \texttt{infer}(\Delta_0' \cdots \Delta_n', [x^n \overset{n}{\mapsto} w]\, e, S_1\beta, \mathcal{Q}_2)$ in
let $S_3 = \texttt{unify}(\{S_2 S_1 A = S_2 S_1 \alpha \to S_2 S_1 \beta\}, \mathcal{Q}_3)$ in $S_3 S_2 S_1$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \texttt{ref}\ e, A, \{\alpha\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2) =$ (IREF)
let $S_1 = \texttt{infer}(\Delta_0 \cdots \Delta_n, e, \alpha, \mathcal{Q}_1)$ in
let $S_2 = \texttt{unify}(\{S_1 A = (S_1\alpha)\ \texttt{ref}\}, \mathcal{Q}_2)$ in $S_2 S_1$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, !\ e, A, \mathcal{Q}) =$ (IDEREF)
$\texttt{infer}(\Delta_0 \cdots \Delta_n, e, A\ \texttt{ref}, \mathcal{Q})$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, e_1 := e_2, A, \mathcal{Q}_1 \oplus \mathcal{Q}_2) =$ (IASSIGN)
let $S_1 = \texttt{infer}(\Delta_0 \cdots \Delta_n, e_1, A\ \texttt{ref}, \mathcal{Q}_1)$ in
let $S_2 = \texttt{infer}((S_1\Delta_0) \cdots (S_1\Delta_n), e_2, S_1 A, \mathcal{Q}_2)$ in $S_2 S_1$

$\texttt{infer}(\Delta_0 \cdots \Delta_n, \texttt{let}\ (x\ e_1)\ e_2, A, \{\alpha\} \oplus \mathcal{Q}_1 \oplus \mathcal{Q}_2 \oplus \mathcal{Q}_3) =$ (ILET)
if $\mathrm{expansive}^n(e_1)$
then
let $S_1 = \texttt{infer}(\Delta_0 \cdots \Delta_n, e_1, \alpha, \mathcal{Q}_1)$ in
let $(S_2, \Delta_0' \cdots \Delta_n') = \texttt{add}((S_1\Delta_0) \cdots (S_1\Delta_n), \{x : S_1\alpha\}, \mathcal{Q}_2)$ in
let $S_3 = \texttt{infer}(\Delta_0' \cdots \Delta_n', e_2, S_2 S_1 A, \mathcal{Q}_3)$ in $S_3 S_2 S_1$
else
let $S_1 = \texttt{infer}(\Delta_0 \cdots \Delta_n, e_1, \alpha, \mathcal{Q}_1)$ in
let $\{\xi_1, \ldots, \xi_m\} = \mathrm{FV}(S_1\alpha) \setminus \bigcup_{i=0}^n \mathrm{FV}(S_1\Delta_i)$
let $(S_2, \Delta_0' \cdots \Delta_n') =$
$\texttt{add}((S_1\Delta_0) \cdots (S_1\Delta_n), \{x : \forall \xi_1 \ldots \xi_m . S_1\alpha\}, \mathcal{Q}_2)$ in
let $S_3 = \texttt{infer}(\Delta_0' \cdots \Delta_n', e_2, S_2 S_1 A, \mathcal{Q}_3)$ in $S_3 S_2 S_1$

Figure 10. Type inference algorithm for λ^{poly}_{open}

PROOF By induction on the type derivation of $\Gamma_0 \cdots \Gamma_n \vdash^i e : A$ in the implicit $\lambda^{\square}$ language. □

6.2 Relation to λ^{i}_{let}

λ^{i}_{let} [3] is not embedded in λ^{poly}_{open} while its monomorphic version λ^{i} [3] is embedded in λ^{sim}_{open} if its cross-stage persistence operator (%) is removed. Figure 13 shows the type system of λ^{i}_{let}. We omit the type system of λ^{i}, which is just a simply typed version of λ^{i}_{let} and does not have expression $\texttt{let}\ (x\ e_1)\ e_2$.

$$
\begin{array}{l}
\mathtt{add}(\Delta_0 \cdots \Delta_n, \{x:\tau\}, \{\theta, \rho'\}) = \\
\quad \text{if } \Delta_n = \{x:\mu\} :: \Delta' \text{ then} \\
\qquad (\,\varnothing,\ \Delta_0 \cdots \Delta_{n-1}\,(\{x:\tau\} :: \Delta')\,) \\
\quad \text{else if } \Delta_n = \{x_i:\mu_i\}_1^k \text{ and } x \notin \{x_i\}_1^k \text{ then} \\
\qquad (\,\varnothing,\ \Delta_0 \cdots \Delta_{n-1}\,(\{x:\tau\} :: \{x_i:\mu_i\}_1^k)\,) \\
\quad \text{else} \\
\qquad \text{let } \Delta_n = \{x_i:\mu_i\}_1^k :: \rho \text{ and } x \notin \{x_i\}_1^k \text{ in} \\
\qquad \text{let } S = \{\rho : \{x:\theta\} :: \rho'\} \text{ in} \\
\qquad\quad (\,S,\ (S\Delta_0) \cdots (S\Delta_{n-1})\,(\{x:S\tau\} :: \{x_i:S\mu_i\}_1^k :: \rho')\,)
\end{array}
$$

Figure 11. Extension of type environments

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & x \mid \lambda x.e \mid e\,e \mid \mathtt{box}\,e \mid \mathtt{unbox}_k\,e \\
\text{Types} & A, B & ::= & \iota \mid A \to B \mid \Box A \\
\text{Type Environments} & \Gamma & ::= & \varnothing \mid \Gamma + x : A
\end{array}
$$

$$
\frac{\Gamma_n(x) = A}{\Gamma_0 \cdots \Gamma_n \vdash^i x : A} \qquad \frac{\Gamma_0 \cdots \Gamma_n + x : A \vdash^i e : B}{\Gamma_0 \cdots \Gamma_n \vdash^i \lambda x.e : A \to B}
$$

$$
\frac{\Gamma_0 \cdots \Gamma_n \vdash^i e_1 : A \to B \quad \Gamma_0 \cdots \Gamma_n \vdash^i e_2 : A}{\Gamma_0 \cdots \Gamma_n \vdash^i e_1 e_2 : B}
$$

$$
\frac{\Gamma_0 \cdots \Gamma_n \varnothing \vdash^i e : A}{\Gamma_0 \cdots \Gamma_n \vdash^i \mathtt{box}\,e : \Box A} \qquad \frac{\Gamma_0 \cdots \Gamma_n \vdash^i e : \Box A}{\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash^i \mathtt{unbox}_k\,e : A}
$$

Figure 12. Type system for the implicit $\lambda^{\Box}$

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & x \mid \lambda x.e \mid e_1 e_2 \\
 & & & \mid \langle e \rangle \mid \tilde{}\,e \mid \%\,e \mid \mathtt{run}\,e \\
 & & & \mid \mathtt{open}\,e \mid \mathtt{close}\,e \\
 & & & \mid \mathtt{let}\,(x\,e_1)\,e_2 \\
\text{Types} & A, B & ::= & \iota \mid A \to B \mid \langle A \rangle^{\alpha} \mid \langle A \rangle \\
\text{Type Schemes} & \tau & ::= & A \mid \forall \kappa.\tau \\
\text{Type Environments} & \Gamma & ::= & \varnothing \mid \Gamma + x : A^L \\
\text{Type Scheme Environments} & \Delta & ::= & \varnothing \mid \Delta + x : \tau^L
\end{array}
$$

$$
\frac{\Delta(x) = \tau^L \quad \tau \succ A}{\Delta \vdash^L x : A} \qquad \frac{\Delta + x : A^L \vdash^L e : B}{\Delta \vdash^L \lambda x.e : A \to B}
$$

$$
\frac{\Delta \vdash^L e_1 : A \to B \quad \Delta \vdash^L e_2 : A}{\Delta \vdash^L e_1 e_2 : B}
$$

$$
\frac{\Delta \vdash^{L\alpha} e : A}{\Delta \vdash^L \langle e \rangle : \langle A \rangle^{\alpha}} \qquad \frac{\Delta \vdash^L e : \langle A \rangle^{\alpha}}{\Delta \vdash^{L\alpha} \tilde{}\,e : A}
$$

$$
\frac{\Delta \vdash^L e : \langle A \rangle}{\Delta \vdash^L \mathtt{run}\,e : A} \qquad \frac{\Delta \vdash^L e : A}{\Delta \vdash^{L\alpha} \%\,e : A}
$$

$$
\frac{\Delta \vdash^L e : \langle A \rangle}{\Delta \vdash^L \mathtt{open}\,e : \langle A \rangle^{\alpha}} \qquad \frac{\Delta \vdash^L e : \langle A \rangle^{\alpha} \quad \alpha \notin \mathrm{FV}(\Delta, L, A)}{\Delta \vdash^L \mathtt{close}\,e : \langle A \rangle}
$$

$$
\frac{\begin{array}{l}\Delta \vdash^L e_1 : A \\ \Delta + x : \forall \kappa_1 \ldots \kappa_m.A^L \vdash^L e_2 : B \quad \{\kappa_i\}_1^m = \mathrm{FV}(A) \setminus \mathrm{FV}(\Delta, L)\end{array}}{\Delta \vdash^L \mathtt{let}\,(x\,e_1)\,e_2 : B}
$$

Figure 13. Type system for λ^i_{let}

Expression translation from λ^i without % to λ^{sim}_{open} is straightforward. Because λ^i preserves alpha-equivalence, we assume without loss of generality that all bound variables are distinct in λ^i before execution.

$$
\begin{array}{rlrl}
[\![x]\!] &= x & [\![\lambda x.e]\!] &= \lambda^* x.[\![e]\!] \\
[\![e_1 e_2]\!] &= [\![e_1]\!][\![e_2]\!] & [\![\langle e \rangle]\!] &= \mathtt{box}\ [\![e]\!] \\
[\![\tilde{}\,e]\!] &= \mathtt{unbox}[\![e]\!] & [\![\mathtt{run}\ e]\!] &= \mathtt{unbox}_0[\![e]\!] \\
[\![\mathtt{open}\ e]\!] &= \mathtt{open}\ [\![e]\!] & [\![\mathtt{close}\ e]\!] &= [\![e]\!]
\end{array}
$$

Note that because runtime renaming that happens implicitly in λ^i is simulated by our hygienic abstraction $\lambda^* x.e$, the translated version in λ^{sim}_{open} has the same semantics as the original one in λ^i.

DEFINITION 6.1. *(Collecting Type Environments) Let* $\{\Gamma_i \vdash^{L_i} e_i : A_i\}_1^m$ *be a set of typing judgments occurring in* λ^i*'s type derivation tree of* $\Gamma \vdash^L e : A$

$$\mathrm{CTE}(\Gamma \vdash^L e : A) = \bigcup_{i=1}^m \Gamma_i.$$

As we assume that all bound variables are distinct, if $x : A_1^{L_1}$ and $x : A_2^{L_2}$ are in $\mathrm{CTE}(\Gamma \vdash^L e : A)$, then $A_1^{L_1} = A_2^{L_2}$. In other words, $\mathrm{CTE}(\Gamma \vdash^L e : A)$ is always a function. Let Φ be a function satisfying $\Phi \supseteq \mathrm{CTE}(\Gamma \vdash^L e : A)$. Then, type translation from λ^i without % to λ^{sim}_{open} is:

$$
\begin{array}{rcl}
\mathtt{Tr}(\Phi, \iota, L) &=& \iota \\
\mathtt{Tr}(\Phi, A \to B, L) &=& \mathtt{Tr}(\Phi, A, L) \to \mathtt{Tr}(\Phi, B, L) \\
\mathtt{Tr}(\Phi, \langle A \rangle^{\alpha}, L) &=& \Box([\![\Phi]\!]^{L\alpha}_{|L\alpha|} \triangleright \mathtt{Tr}(\Phi, A, L\alpha)) \\
\mathtt{Tr}(\Phi, \langle A \rangle, L) &=& \Box(\varnothing \triangleright \mathtt{Tr}(\Phi, A, L')) \\
 & & \textit{where } L' \textit{ is any stage,}
\end{array}
$$

and environment translation is:

$$
[\![\Phi]\!]^L_i = \{x : \mathtt{Tr}(\Phi, A, L') \mid L' \textit{ is a prefix of } L,\ |L'| = i,\ x : A^{L'} \in \Phi\}
$$

LEMMA 6.2. *If* $\Gamma \vdash^L e : A$ *in* λ^i *without %, then*

$$[\![\Phi]\!]^L_0 \cdots [\![\Phi]\!]^L_{|L|} \vdash [\![e]\!] : \mathtt{Tr}(\Phi, A, L) \ \textit{ in } \lambda^{\mathrm{poly}}_{\mathrm{open}},$$

for any function $\Phi \supseteq \mathrm{CTE}(\Gamma \vdash^L e : A)$.

PROOF By induction on the type derivation of $\Gamma \vdash^L e : A$ in λ^i without the cross-stage persistence. □

Although translation from λ^i_{let} to λ^{poly}_{open} seems straightforward, λ^i_{let} is not conservatively embedded in λ^{poly}_{open}. The code template type of $\langle e \rangle$ is not always translated into some modal type of $\mathtt{box}\ e$. For example, code template

$$\langle\ \text{let } x = \lambda y.y \text{ in } \tilde{}\langle xx \rangle\ \rangle$$

is admissible to λ^i_{let}'s type system as follows.

$$
\dfrac{\dfrac{y : B^{\alpha} \vdash^{\alpha} y : B}{\varnothing \vdash^{\alpha} \lambda y.y : B \to B} \qquad \dfrac{\dfrac{x : \forall \kappa.\kappa \to \kappa^{\alpha} \vdash^{\alpha} xx : A \to A}{x : \forall \kappa.\kappa \to \kappa^{\alpha} \vdash^{\varepsilon} \langle xx \rangle : \langle A \to A \rangle^{\alpha}}}{x : \forall \kappa.\kappa \to \kappa^{\alpha} \vdash^{\alpha} \tilde{}\langle xx \rangle : A \to A}}{\dfrac{\varnothing \vdash^{\alpha} \text{let } x = \lambda y.y \text{ in } \tilde{}\langle xx \rangle : A \to A}{\varnothing \vdash^{\varepsilon} \langle\ \text{let } x = \lambda y.y \text{ in } \tilde{}\langle xx \rangle\ \rangle : \langle A \to A \rangle^{\alpha}}}
$$

In the above type derivation tree, code template $\langle xx \rangle$ is typable in λ^i_{let}. However, code template $\mathtt{box}\ (xx)$ is not typable in λ^{poly}_{open} because λ^{poly}_{open} allows only monomorphic type environment inside the code template types (i.e., $\Box(\Gamma \triangleright A)$ not $\Box(\Delta \triangleright A)$). Note that xx is not typable under any monomorphic type environment.

On the other hand, free variables of $\langle e \rangle$ should be lexically bound at the same stage level, which makes it difficult to support imperative operations for open code templates. For example, suppose that open code template $\langle x \rangle$ is stored at a stage L under some Δ where $x : A^{L\alpha} \in \Delta$. The stored open code template may be later dereferenced at a stage L under some Δ' ($x : A^{L\alpha} \notin \Delta'$). This example is typable in λ^{poly}_{open} and not in λ^i_{let}.

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

$\lambda^{\bigcirc}$ [6] is embedded in λ^i [3], and does not have the cross-stage persistence operator (%). Because λ^i without % is embedded in λ^{sim}_{open} (Section 6.2), $\lambda^{\bigcirc}$ is thus embedded in λ^{sim}_{open}.

6.4 Relation to λ^{+}_{code}

Unlike ours, λ^{+}_{code} [4]'s polymorphic generalization is not allowed inside code templates, and no free named variables can occur inside code templates.

λ^{+}_{code}'s use of deBruijn indices for program variables conflicts with imperative multi-staged programming practice. When an open code template escapes from its lexical scope by some imperative operations, its free variables' deBruijn indices can denote different variables from those in the original program with named variables. For example, consider the following code that is admissible to λ^{+}_{code}'s type system:

```
let val a = ref ‘1
  val f = ‘(fn x -> fn y -> ,(a := ‘(x + y); ‘2))
  val g = ‘(fn y -> fn z -> ,(!a))
in ... end
```

When the stored open code `‘(x+y)` is plugged inside `‘(fn y -> fn z -> ,(!a))`, the programmer intends that the `x` remains free and `y` is bound to *y*. If we replace every variable by its deBruijn index, the resulting program becomes a completely different one. The `x` and `y` in `’(x+y)` will have deBruijn indices 2 and 1, respectively. Thus in `‘(fn y -> fn z -> ,(!a))`, the `x` is bound with *y* and the `y` with *z*.

7. Conclusion

We have presented a polymorphic modal type system and its principal type inference algorithm that conservatively extend ML by all of Lisp's staging constructs (the quasi-quotation system). Our type system supports open code, unrestricted operations on references, intentional variable-capturing substitution as well as capture-avoiding substitution of free variables in open code, and lifting values into code, whose combination escaped all the previous systems.

Type-checked programs preserve alpha-equivalence only at stage 0. This may be unacceptable in a purely functional language, but it is frequently practiced in Lisp's staged programming, hence, we think worthwhile to support it in our type system.

For programs that are accepted by existing multi-staged type systems such as $\lambda^{\Box}$ [7, 8], $\lambda^{\bigcirc}$ [6], and λ^{i} without its cross-stage persistence [3], there exist their semantics-preserving, translated versions that are accepted by our type system.

Acknowledgments

We thank the anonymous referees and Stephanie Weirich for their helpful comments that improved the presentation of this article.

References

[1] D. Ancona and E. Moggi. A fresh calculus for name management. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, October 2004.

[2] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(3), 2003.

[3] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-like inference for classifiers. In *Proceedings of the European Symposium on Programming 2004*, pages 79–93. Springer, 2004.

[4] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *Proceedings of the International Conference on Functional Programming (ICFP '02)*, pages 275–286. ACM, August 2003.

[5] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 242–257. ACM, Jan 1996.

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

[7] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '96)*, pages 258–270. ACM, 1996.

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

[9] Dawson R. Engler. VCODE:A retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 160–170, New York, 1996. ACM.

[10] Paul Graham. *On Lisp: an advanced techniques for Common Lisp*. Prentice Hall, 1994.

[11] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

[13] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. Technical Report ROPAS-2005-26, (ropas.snu.ac.kr/lib/dock/KiYiCa2005.pdf), 2005.

[14] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and functional programming*, pages 151–161. ACM, August 1986.

[15] M. Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 137–148. ACM Press, June 1996.

[16] H. Massalim. *An Efficient Implementation of Functional Operating System Services*. PhD thesis, Columbia University, 1992.

[17] Aleksandar Nanevski. Meta-programming with names and necessity. In *Proceedings of the International Conference on Functional Programming (ICFP '02)*, pages 206–217. ACM, October 2002.

[18] Aleksandar Nanevski and Frank Pfenning. Staged computation with names and necessity. *to appear in Journal of Functional Programming*.

[19] Massimilian Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kasshoek. C and tcc:a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21:324–369, March 1999.

[20] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993.

[21] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[22] Morten Rhiger. First-class open and closed code fragments. In *Proceedings of the Sixth Symposium on Trends in Functional Programming*, September 2005.

[23] Guy L. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.

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

[25] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '03)*. ACM, 2003.

[26] J. B. Wells. The essence of principal typings. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 913–925. Springer-Verlag, 2002.

[27] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, Dec 1995.