Static Analysis of Multi-Staged Programs via Unstaging Translation

Wontae Choi
Seoul National University

Baris Aktemur
UIUC & Ozyegin University

Kwangkeun Yi
Seoul National University

Makoto Tatsuta
National Institute of Informatics

Abstract

Static analysis of multi-staged programs is challenging because the basic assumption of conventional static analysis no longer holds: the program text itself is no longer a fixed static entity, but rather a dynamically constructed value. This article presents a semantic-preserving translation of multi-staged call-by-value programs into unstaged programs and a static analysis framework based on this translation. The translation is semantic-preserving in that every small-step reduction of a multi-staged program is simulated by the evaluation of its unstaged version. Thanks to this translation we can analyze multi-staged programs with existing static analysis techniques that have been developed for conventional unstaged programs: we first apply the unstaging translation, then we apply conventional static analysis to the unstaged version, and finally we cast the analysis results back in terms of the original staged program. Our translation handles staging constructs that have been evolved to be useful in practice (typified in Lisp's quasi-quotation): open code as values, unrestricted operations on references and intentional variable-capturing substitutions. This article omits references for which we refer the reader to our companion technical report.

1. Introduction

Staged programming, which explicitly divides a computation into separate stages, is a unifying principle for the existing program-generation systems. Partial evaluation [22, 13], runtime code generation [17, 33, 27, 29], function inlining, and macro expansion [35, 19], are all instances of staged computation.

There can be arbitrarily many stage levels, 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. A program of stage 1 can be brought to stage 0 for exececution.

The key aspect of multi-staged languages is to have code templates (program fragments) as first-class values. Code templates are freely passed, stored, composed with code of other stages, and executed when appropriate. For this reason, multi-staged programming is also called "meta-programming."

Multi-staged programming is commonplace in mainstream programming. Lisp(or Scheme)'s quasi-quotation system [35, 19] is a fully fledged multi-staged system that has been evolved to comply with the demands from multi-staged programming practices. C's macros and C++'s templates are multi-staged features. C#, JavaScript, PHP, and Python support a form of multi-staged programming, albeit a limited one. MetaOcaml and Template Haskell are extensions to ML and Haskell to support multi-staged programming.

However, static analysis of multi-staged programs (in order to, for example, find bugs or optimize) is mostly unexplored. Aside from static typing systems such as [26, 5, 14, 32, 4, 36], there are, as far as we know, no studies on more general and more powerful semantic-based static analysis (*á la* abstract interpretation) for multi-staged programs.

The primary obstacle is the fact that the basic assumption of conventional static analysis no longer holds: the program text itself is no longer a fixed static entity, but rather a dynamically constructed value. Conventional static analysis can finitely estimate the set of constructed code fragments, but we reach a stalemate after that. If the program executes the generated code, how can we statically analyze the execution? The program text to analyze at this stage is not a usual text but a finitely abstracted representation of the possibly infinite set of generated code.

Contribution

- As a solution to the problem, we present a semantic-preserving translation of multi-staged call-by-value programs into unstaged programs and a static analysis framework based on this translation. We prove the translation is semantic-preserving in that every small-step reduction of a multi-staged program is simulated by the evaluation of its unstaged version.

  Thanks to this translation we can analyze multi-staged programs with existing static analysis techniques that have been developed for conventional unstaged programs: we first apply the unstaging translation, then we apply conventional static analysis to the unstaged version, and finally we cast the analysis results back in terms of the original staged program.

- We present a framework of safely projecting the static analysis results of unstaged translated version back in terms of the original staged program. Once the projection is checked safe, we can use conventional static analysis for the unstaged record language to achieve a static analysis for the multi-staged language.
- Our semantic-preserving translation handles staging constructs that have been evolved to be useful in practice (typified in Lisp's quasi-quotation): open code as values, unrestricted operations on references, and intentional variable-capturing substitutions. This article omits references, for which we refer the reader to our companion technical report.

We illustrate the problem and our solution using an example program. In the example, we use Lisp's quasi-quote syntax [35] for staging constructs.

1.1 Problem

For example, consider the following two-staged program.

$$\begin{array}{l} x \texttt{ := `0;} \\ \texttt{repeat} \\ \quad x \texttt{ := `(,}x\texttt{ + 2)} \\ \texttt{until } cond\texttt{;} \\ \texttt{run } x \end{array}$$

Variable x initially has code `` `0``. The repeat statement repeatedly assigns a new code value to x. The expression `` `(,``x`` + 2)`` becomes a code value by plugging x's current contents into the place of "$,x$". Thus after one iteration x contains `` `(0 + 2)``, after two iterations `` `(0 + 2 + 2)``, and so on. Finally `run` x evaluates the x's code and returns a non-negative even integer.

Now consider statically estimating the above program. In order to estimate the value of the "`run` x" expression we must estimate the set of possible code values that may be assigned to x. Suppose that the number of iterations of the repeat statement is statically undecidable. Then flow-insensitive static analysis, for example, must somehow finitely estimate the set of all possible, infinitely many code values

$$\texttt{`0, `(0+2), `(0+2+2)}, \cdots\}.$$

To finitely approximate the infinite set of code, suppose we use grammar-based abstraction [8, 30, 6]. Then the set of code for `x` would be approximated by a grammar:

$$S \rightarrow \texttt{0} \mid S\texttt{+2}.$$

However, in order to analyze the code run by the "`run` x" expression (at least by conventional analyses for unstaged programs), every code implied by the grammar must be exposed first; that is, the grammar must be concretized. Since the concrete image has infinitely many code values, such analysis is unrealizable. A different static analysis technique that can evade such concretization trap is necessary.

1.2 Solution

As a solution to the problem, we present a three-step approach: translate, analyze, and project. To make this three-step approach correct, we prove the translation semantic-preserving: the translated unstaged version simulates every evaluation step of the original staged program. And we show a sound condition for the projection to be correct, i.e., to be aligned with the correspondence induced by the translation.

Here we will demonstrate these steps with the motivating example just presented. Exact definitions, lemmas and theorems are presented in Sections 2, 3 and 4.

- **Translation**: The above example program is translated as

$$\begin{array}{l} x \texttt{ := } \lambda\rho\texttt{.0;} \\ \texttt{repeat} \\ \quad x \texttt{ := (}\lambda h\texttt{.(}\lambda\rho\texttt{.(}h\ \rho\texttt{)+2)) } x \\ \texttt{until } cond\texttt{;} \\ \texttt{(}x\ \{\}\texttt{)} \end{array}$$

The translation works as follows.

  - Code is translated into a function that explicitly takes a record (for its environment) as an argument:

$$\texttt{`0} \longmapsto \lambda\rho\texttt{.0}$$

    Hence, the run expression is translated into a function application:

$$\texttt{run `0} \longmapsto \texttt{(}\lambda\rho\texttt{.0)}\{\}$$

    The function is applied to an empty record because the code `` `0`` has no free variable.
  - Free variables inside a code are translated to record access expressions. For example,

$$\texttt{`}x \longmapsto \lambda\rho\texttt{.}\rho\cdot x$$

  - Code composition `` `(,``x`` + 2)`` is translated to a function(for the resulting code)-generating application whose actual parameter is the part for the code-to-be-plugged expression:

$$\texttt{`(,}x\texttt{ + 2)} \longmapsto \texttt{(}\lambda h\texttt{.(}\lambda\rho\texttt{.(}h\ \rho\texttt{)+2)) } x$$

    The code value of x will be plugged into its corresponding hole (the place of "h"). The "$\lambda\rho$`.(`$h\ \rho$`)+2`" stands for the resulting code. The application "`(`$h\ \rho$`)`" is for capturing the code's, if any, free variables by the current environment.
  - The evaluation of the unstaged version simulates that of the original staged program. For example, after one iteration of the repeat statement, x has $\lambda\rho$`.((`ρ`.0)`ρ`)+2`. After two iterations, $\lambda\rho$`.((`$\lambda\rho$`.((`$\lambda\rho$`.0)`ρ`)+2)`ρ`)+2`, and so on. These functions correspond to code values `` `(0+2)`` and `` `(0+2+2)`` after the same numbers of iterations in the original staged program.

- **Analysis**: Because the translation removes all the staging features, we can apply conventional static analysis techniques to translated results.

  For example, suppose we estimate the values of expressions by a simple flow-insensitive value analysis with 0CFA. (We can apply any elaborate static analysis technique, but just for illustration this simple analysis is sufficient.) We present the analysis results in set-constraint style [20, 21]. We write V_i or $V_{\mathtt{x}}$ for the values of expression i and variable x respectively. Let us first label some expressions including lambdas:

$$\begin{array}{l} x \texttt{ := } \lambda\rho_1\texttt{.0;} \\ \texttt{repeat} \\ \quad x \texttt{ := (}\lambda h\texttt{.(}\lambda\rho_2\texttt{.}\underline{\texttt{(}h\ \rho_2\texttt{)}_1}\texttt{ + 2)) } x \\ \texttt{until } cond\texttt{;} \\ \underline{\texttt{(}x\ \{\}\texttt{)}_2} \end{array}$$

The analysis will deduce set constraints as follows. For brevity, we write "$\lambda\rho_i$" omitting the body expression for lambdas (0CFA-closure values).

From the first assignment statement,

$$V_x \ni \lambda\rho_1.$$

From the assignment inside the repeat statement, V_x can also contain the value of the application "`(`λ`h...) x`", i.e., λ`h`'s body expression's value, which is $\lambda\rho_2$. Hence

$$V_x \ni \lambda\rho_2.$$

The parameter binding in the application "`(λh....) x`" gives

$$V_h \ni V_x, \quad \text{hence} \quad V_h \ni \lambda\rho_1 \text{ and } V_h \ni \lambda\rho_2.$$

Application expression "$\underline{(\mathtt{h}\ \rho_2)}_1$" has values of called functions' body expressions. The called functions would be V_h, which has $\lambda\rho_1$ and $\lambda\rho_2$. Thus,

$V_1 \ni \mathtt{0}$ (* V_h has $\lambda\rho_1$ and $\lambda\rho_1$'s body's value is 0 *)
$V_1 \ni V_1\mathtt{+2}.$ (* V_h has $\lambda\rho_2$ and $\lambda\rho_2$'s body's value is V_1+2 *)

Similarly, from the application expression "$\underline{(x\ \{\})}_2$",

$V_2 \ni \mathtt{0}$ (* V_x has $\lambda\rho_1$ and $\lambda\rho_1$'s body's value is 0 *)
$V_2 \ni V_1\mathtt{+2}.$ (* V_x has $\lambda\rho_2$ and $\lambda\rho_2$'s body's value is V_1+2 *)

The above constraints can be understood as inductive rules for value sets. For example, the (infinite) sets V_1 and V_2 are inductively defined as follows:

$$V_1 \to \mathtt{0} \mid V_1\mathtt{+2}$$
$$V_2 \to \mathtt{0} \mid V_1\mathtt{+2}$$

Thus we can conclude that V_1 and V_2 consist of all non-negative even integers.

- **Projection**: Finally, the analysis results for the unstaged version need to be cast back in terms of the original staged program. Because code (backquote) expressions are translated into lambdas, some lambdas in the above example analysis' results correspond to the code expressions in the original staged program. For example, analysis result V_h for variable h has $\lambda\rho_1$ and $\lambda\rho_2$ which respectively corresponds to code expressions `` `0 `` and `` `(, ``x`` + 2)``. That is, code to be plugged into the place of "`,`x" can be `` `0 `` and, recursively, `` `(, ``x`` + 2)``.

  It is straightforward to keep track of which lambdas in the unstaged version correspond to which code expression in the staged original. We can, for instance, assign parameter names of such lambdas from a unique namespace to identify the corresponding code expression, such as, $\lambda\rho_i$ for the lambda translated from code expression `` ` ``e_i of index i.

  Regarding the projections, we cannot use arbitrary ones. Arbitrary projections of the static analysis results of the translated program can have images that fail to qualify as static analysis results of the original staged program. Projection from abstract semantics of the translated program to that of the subject program must be a safe approximation of its concrete counterpart (projection from concrete semantics of the translated program to that of the subject program). Section 4 presents the formalization of this condition and an analysis example.

Comparisons

- Translation: Davies and Pfenning's unstaging translation [14] works only for closed code. Their translation does not support open code and intentional variable-capturing substitution at stages > 0 ("unhygienic" macros). This feature, which may be unacceptable in a purely functional language, has long been used in practice (for example by Lisp's quasi-quote programmers) for efficiency programming convenience. Kameyama et.al [24]'s translation supports open code but they do not provide an observational equivalence; hence it is not adequate for our purpose: a round-about static analysis approach for multi-staged programs.

  Our unstaging translation is a refinement of [1, 2]. We prove only two kinds of administrative reductions suffice whose exhaustive application reaches the admin-normal form. We also define an inverse translation that converts expressions in the admin-normal form back to the original staged expression.

- Static analysis: Most static analyses for multi-staged programs are string analyses for programs that generate code as strings, but they are limited to estimate only the shape, not the semantics, of generated code by using a grammar [8, 30, 6] or the parsing stack [16]. Such string analyses do not analyze the semantics of the generated code string.

  Multi-staged static type systems [14, 36, 26, 39] and their inference algorithms can be considered sound static analyses, but extending them for analyzing other behavior than types (*á la* effect type systems [28, 23, 38]) is also constrained by the aforementioned infinite-concretization trap. Any extension to estimate other properties than types is limited to those that can proceed without analyzing the semantics of the generated code. Existing multi-staged static type systems can evade the infinite concretization trap because typing the execution of the generated code (for expression such as `run` e) does not have to analyze the generated code itself but can just pick up the type from the generated code's type.

1.3 Organization

Section 2 defines the subject call-by-value multi-staged language $\lambda_{\mathcal{S}}$ and the target unstaged record language $\lambda_{\mathcal{R}}$. Section 3 defines and proves semantic-preservation of the unstaging translation from $\lambda_{\mathcal{S}}$ to $\lambda_{\mathcal{R}}$. Section 4 presents a condition for safe projection. Section 5 discusses related works. Section 6 concludes.

2. Languages

In this section we give the formal definitions of the subject staged language $\lambda_{\mathcal{S}}$ and the target record language $\lambda_{\mathcal{R}}$. For each, we present the syntax, operational semantics and the type system.

2.1 Multi-Staged Language $\lambda_{\mathcal{S}}$

The language $\lambda_{\mathcal{S}}$ is a typed, call-by-value λ-calculus with staging annotations. It is based on λ^{sim}_{open} [26], simplified by removing hygienic code composition (i.e. λ^*), mutable reference, and the lift operation. Also, the `unbox` operator is restricted to 1 stage. In this work our focus is not on polymorphism. Thus, we omit let-bindings from the syntax; we use them in the examples as a syntactic sugar for application.

Syntax

$$\mathit{Variable} \quad x, y, f \in \mathit{Var}_{\mathcal{S}}$$

$$\begin{array}{lrl} \mathit{Expr}_{\mathcal{S}} & e ::= & i \mid x \mid \lambda x.e \mid e\ e \mid \mathtt{fix}\ f x.e \\ & & \mid \mathtt{box}\ e \mid \mathtt{unbox}\ e \mid \mathtt{run}\ e \end{array}$$

The syntax of $\lambda_{\mathcal{S}}$ is given above. The language contains constants, variables, lambda abstraction, application, and the fixpoint operator `fix`. Finally, there are staging annotations: `box` is used to define code templates. `unbox` is the escape operator that defines a "hole" inside a code template which is filled in with another code template. `box` and `unbox` operators can be arbitrarily nested. `run` executes a code template.

Operational Semantics

$\lambda_{\mathcal{S}}$ has a small-step, call-by-value, operational semantics. Evaluation rules of the language are in Figure 1. The evaluation $e \xrightarrow{n} e'$ has the meaning that "the expression e is evaluated to e' at stage n."

Values are expressions that cannot be reduced further. Values are defined for all stages. At stage 0, values are constants, functions and code templates. A code template is a frozen expression within a `box` annotation. Inside code templates, holes denoted by the `unbox` are filled in by evaluating the unboxed expression to a code template. In other words, code templates are composed using the `unbox` operator. Only stage-1 holes can be filled in. Once all the holes

Definitions

$$\begin{array}{lll} \mathit{Value}^0 & v^0 ::= i \mid \lambda x.e \mid \mathtt{fix}\ fx.e \mid \mathtt{box}\ v^1 \\ \mathit{Value}^n\ (n>0) & v^n ::= i \mid x \mid \lambda x.v^n \mid v^n v^n \mid \mathtt{fix}\ fx.v^n \\ & \quad \mid \mathtt{box}\ v^{n+1} \mid \mathtt{unbox}\ v^{n-1}\ (n>1) \\ & \quad \mid \mathtt{run}\ v^n \end{array}$$

Operational Semantics $(n \geq 0)$

$$\text{(APP)} \quad \frac{e_1 \xrightarrow{n} e_1'}{e_1\ e_2 \xrightarrow{n} e_1'\ e_2} \qquad \frac{e \xrightarrow{n} e' \quad v \in \mathit{Value}^n}{v\ e \xrightarrow{n} v\ e'}$$

$$(\lambda x.e)\ v \xrightarrow{0} [x \overset{0}{\mapsto} v]e$$

$$(\mathtt{fix}\ fx.e)\ v \xrightarrow{0} [x \overset{0}{\mapsto} v][f \overset{0}{\mapsto} \mathtt{fix}\ fx.e]e$$

$$\text{(BOX)} \quad \frac{e \xrightarrow{n+1} e'}{\mathtt{box}\ e \xrightarrow{n} \mathtt{box}\ e'}$$

$$\text{(RUN)} \quad \frac{e \xrightarrow{n} e'}{\mathtt{run}\ e \xrightarrow{n} \mathtt{run}\ e'} \qquad \frac{v \in \mathit{Value}^1}{\mathtt{run}\ (\mathtt{box}\ v) \xrightarrow{0} v}$$

$$\text{(UNB)} \quad \frac{e \xrightarrow{n} e'}{\mathtt{unbox}\ e \xrightarrow{n+1} \mathtt{unbox}\ e'} \qquad \frac{v \in \mathit{Value}^1}{\mathtt{unbox}\ (\mathtt{box}\ v) \xrightarrow{1} v}$$

$$\text{(ABS)} \quad \frac{e \xrightarrow{n+1} e'}{\lambda x.e \xrightarrow{n+1} \lambda x.e'}$$

$$\text{(FIX)} \quad \frac{e \xrightarrow{n+1} e'}{\mathtt{fix}\ fx.e \xrightarrow{n+1} \mathtt{fix}\ fx.e'}$$

Figure 1. Operational Semantics of $\lambda_{\mathcal{S}}$.

are filled, a code template becomes a box-value. A code template can be evaluated at stage 0 by `run`.

$\lambda_{\mathcal{S}}$ extends lambda calculus conservatively. At stage 0, (APP) is same as the traditional substitution-based call-by-value semantics. Alpha conversion and beta reduction are available at stage-0.

Type System

Figure 2 shows a monomorphic type system for $\lambda_{\mathcal{S}}$. A polymorphic type system is also available [26]. Types in $\lambda_{\mathcal{S}}$ are defined as below.

$$\begin{array}{lll} \mathit{Types}_{\mathcal{S}} & \mathtt{T} & ::= \iota \mid \mathtt{T} \rightarrow \mathtt{T} \mid \Box(\Gamma \rhd \mathtt{T}) \\ \mathit{Type\ Environments}_{\mathcal{S}} & \Gamma & \in \mathit{Var}_{\mathcal{S}} \xrightarrow{\mathit{fin}} \mathit{Types}_{\mathcal{S}} \end{array}$$

We use $\mathtt{T}$ to denote type terms, ι for base types, $\mathtt{T} \rightarrow \mathtt{T}$ for function types, $\Box(\Gamma \rhd \mathtt{T})$ for code template types, Γ for type environments. A code template is given a box-type $\Box(\Gamma \rhd \mathtt{T})$ with the meaning that "the code template will evaluate to a value of type $\mathtt{T}$ if put in a context that provides the environment Γ." The type environment Γ in the box-type $\Box(\Gamma \rhd \mathtt{T})$ contains the types of the unbound variables in the code template.

A type environment Γ is a mapping from variables to types. $\Gamma + \{x : \mathtt{T}\}$ is a function update operation that defines a function as follows: $(\Gamma + \{x : \mathtt{T}\})(x) = \mathtt{T}$ and $(\Gamma + \{y : \mathtt{T}\})(x) = \Gamma(x)$ if $x \neq y$. A typing judgment has the form $\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} e : \mathtt{T}$ with the meaning that "a stage-n expression e, under type environments $\Gamma_0 \ldots \Gamma_n$, has type $\mathtt{T}$." $\Gamma_0 \ldots \Gamma_n$ is a sequence of type environments. Each type environment corresponds to a stage where Γ_n is the current (i.e. most recent) type environment. For a proof of the soundness of this type system and its let-polymorphic extension, see [26].

$$\text{(CONt)} \quad \Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} i : \iota$$

$$\text{(VARt)} \quad \frac{\Gamma_n(x) = \mathtt{T}}{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} x : \mathtt{T}}$$

$$\text{(ABSt)} \quad \frac{\Gamma_0 \ldots \Gamma_n + \{x : \mathtt{T}_1\} \vdash_{\mathcal{S}} e : \mathtt{T}_2}{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} \lambda x.e : \mathtt{T}_1 \rightarrow \mathtt{T}_2}$$

$$\text{(FIXt)} \quad \frac{\Gamma_0 \ldots \Gamma_n + \{x : \mathtt{T}_1\} + \{f : \mathtt{T}_1 \rightarrow \mathtt{T}_2\} \vdash_{\mathcal{S}} e : \mathtt{T}_2}{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} \mathtt{fix}\ fx.e : \mathtt{T}_1 \rightarrow \mathtt{T}_2}$$

$$\text{(APPt)} \quad \frac{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} e_1 : \mathtt{T}_1 \rightarrow \mathtt{T}_2 \quad \Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} e_2 : \mathtt{T}_1}{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} e_1\ e_2 : \mathtt{T}_2}$$

$$\text{(BOXt)} \quad \frac{\Gamma_0 \ldots \Gamma_n, \Gamma \vdash_{\mathcal{S}} e : \mathtt{T}}{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} \mathtt{box}\ e : \Box(\Gamma \rhd \mathtt{T})}$$

$$\text{(UNBt)} \quad \frac{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} e : \Box(\Gamma_{n+1} \rhd \mathtt{T})}{\Gamma_0 \ldots \Gamma_n, \Gamma_{n+1} \vdash_{\mathcal{S}} \mathtt{unbox}\ e : \mathtt{T}}$$

$$\text{(RUNt)} \quad \frac{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} e : \Box(\varnothing \rhd \mathtt{T})}{\Gamma_0 \ldots \Gamma_n \vdash_{\mathcal{S}} \mathtt{run}\ e : \mathtt{T}}$$

Figure 2. Type System of $\lambda_{\mathcal{S}}$.

2.2 The Record Calculus $\lambda_{\mathcal{R}}$

The language $\lambda_{\mathcal{R}}$ is a λ-calculus with record operations. As the target language of our translation, it is sufficient for the record expression to have only variables and values. As opposed to $\lambda_{\mathcal{S}}$, we include let-bindings in $\lambda_{\mathcal{R}}$. This is to be able to syntactically distinguish several $\lambda_{\mathcal{R}}$ expressions during inverse translation (Section 3.3). The language is still monomorphic.

Syntax

$$\begin{array}{llll} \mathit{Variable} & \rho & \in & \mathit{Var}_P \quad \text{(record variables)} \\ & h & \in & \mathit{Var}_H \quad \text{(hole variables)} \\ & x, y, f & \in & \mathit{Var}_X = \mathit{Var}_{\mathcal{S}} \quad \text{(ordinary variables)} \\ & w & \in & \mathit{Var}_{\mathcal{R}} = \mathit{Var}_X \cup \mathit{Var}_P \cup \mathit{Var}_H \end{array}$$

$$\begin{array}{llll} \mathit{Label} & \mathtt{x} & \in & \mathit{Label} = \{\mathtt{x} \mid x \in \mathit{Var}_X\} \\ & & & \text{(ordinary variables in typewriter font)} \end{array}$$

$$\begin{array}{ll} \mathit{Expr}_{\mathcal{R}} & e ::= i \mid w \mid \lambda w.e \mid e\ e \mid \mathtt{fix}\ fx.e \\ & \quad \mid r \mid r \cdot \mathtt{x} \mid \mathtt{let}\ w = e\ \mathtt{in}\ e \end{array}$$

$$\begin{array}{ll} \mathit{Value}_{\mathcal{R}} & v ::= i \mid \lambda w.e \mid \mathtt{fix}\ fx.e \mid v_r \\ \mathit{Record\ Value}_{\mathcal{R}} & v_r ::= \{\} \mid v_r + \{\mathtt{x} = v\} \end{array}$$

$$\begin{array}{ll} \mathit{Record}_{\mathcal{R}} & r ::= \{\} \mid \rho \mid r + \{\mathtt{x} = x\} \mid r + \{\mathtt{x} = v\} \end{array}$$

The record language $\lambda_{\mathcal{R}}$ has constants (i), variables (x), lambda abstractions, applications, a fixpoint operator `fix`, and let-expressions. As for the record operations there are empty records $(\{\})$, record variables (ρ), and the record update operations $r + \{\mathtt{x} = \_\}$. For field names (or labels) in records, we use variables written in teletype font.

We separate variables into three disjoint sets: ordinary variables Var_X (which are the same as variables of $\lambda_{\mathcal{S}}$), record variables Var_P, and hole variables Var_H. This syntactic distinction makes our presentation of the inverse translation in Section 3.3 easier. The operational semantics does not need to make a distinction; all variables are treated uniformly.

Operational Semantics

$\lambda_{\mathcal{R}}$ has a small-step, call-by-value operational semantics. The evaluation $e \xrightarrow{\mathcal{R}} e'$ means that "the expression $e$evaluates to expres-

Operational Semantics

$$(\mathrm{APP}_{\mathcal{R}})\qquad \frac{e_1 \xrightarrow{\mathcal{R}} e_1'}{e_1\ e_2 \xrightarrow{\mathcal{R}} e_1'\ e_2} \qquad \frac{e \xrightarrow{\mathcal{R}} e'}{v\ e \xrightarrow{\mathcal{R}} v\ e'}$$

$$(\lambda w.e)\ v \xrightarrow{\mathcal{R}} [w \mapsto v]e$$

$$(\mathtt{fix}\ fx.e)\ v \xrightarrow{\mathcal{R}} [x \mapsto v][f \mapsto \mathtt{fix}\ fx.e]e$$

$$(\mathrm{LET}_{\mathcal{R}})\qquad \frac{e_1 \xrightarrow{\mathcal{R}} e_1'}{\mathtt{let}\ \ w = e_1\ \mathtt{in}\ \ e_2 \xrightarrow{\mathcal{R}} \mathtt{let}\ \ w = e_1'\ \mathtt{in}\ \ e_2}$$

$$\mathtt{let}\ \ w = v\ \mathtt{in}\ \ e \xrightarrow{\mathcal{R}} [w \mapsto v]e$$

$$(\mathrm{ACC}_{\mathcal{R}})\qquad v_r \cdot \mathtt{x} \xrightarrow{\mathcal{R}} v_r(\mathtt{x})$$

Record Lookup

$$v_r(\mathtt{x}) = \begin{cases} v & \text{if } v_r = v_r' + \{\mathtt{x} = v\} \\ v_r'(\mathtt{x}) & \text{if } v_r = v_r' + \{\mathtt{y} = \_\} \text{ and } \mathtt{x} \neq \mathtt{y} \end{cases}$$

Figure 3. Operational Semantics of $\lambda_{\mathcal{R}}$.

$$(\mathrm{CONt}_{\mathcal{R}})\qquad \Gamma \vdash_{\mathcal{R}} i : \iota$$

$$(\mathrm{VARt}_{\mathcal{R}})\qquad \frac{\Gamma(w) = \mathtt{T}}{\Gamma \vdash_{\mathcal{R}} w : \mathtt{T}}$$

$$(\mathrm{ABSt}_{\mathcal{R}})\qquad \frac{\Gamma + \{w : \mathtt{T}_1\} \vdash_{\mathcal{R}} e : \mathtt{T}_2}{\Gamma \vdash_{\mathcal{R}} \lambda w.e : \mathtt{T}_1 \to \mathtt{T}_2}$$

$$(\mathrm{FIXt}_{\mathcal{R}})\qquad \frac{\Gamma + \{x : \mathtt{T}_1\} + \{f : \mathtt{T}_1 \to \mathtt{T}_2\} \vdash_{\mathcal{R}} e : \mathtt{T}_2}{\Gamma \vdash_{\mathcal{R}} \mathtt{fix}\ fx.e : \mathtt{T}_1 \to \mathtt{T}_2}$$

$$(\mathrm{APPt}_{\mathcal{R}})\qquad \frac{\Gamma \vdash_{\mathcal{R}} e_1 : \mathtt{T}_1 \to \mathtt{T}_2 \quad \Gamma \vdash_{\mathcal{R}} e_2 : \mathtt{T}_1}{\Gamma \vdash_{\mathcal{R}} e_1\ e_2 : \mathtt{T}_2}$$

$$(\mathrm{LETt}_{\mathcal{R}})\qquad \frac{\Gamma \vdash_{\mathcal{R}} e_1 : \mathtt{T}_1 \quad \Gamma + \{w : \mathtt{T}_1\} \vdash_{\mathcal{R}} e_2 : \mathtt{T}_2}{\Gamma \vdash_{\mathcal{R}} \mathtt{let}\ \ w = e_1\ \mathtt{in}\ \ e_2 : \mathtt{T}_2}$$

$$(\mathrm{EMPt}_{\mathcal{R}})\qquad \Gamma \vdash_{\mathcal{R}} \{\} : \varnothing$$

$$(\mathrm{UPDt}_{\mathcal{R}})\qquad \frac{\Gamma \vdash_{\mathcal{R}} r : \mathtt{T_r} \quad \Gamma \vdash_{\mathcal{R}} e : \mathtt{T}}{\Gamma \vdash_{\mathcal{R}} r + \{\mathtt{x} = e\} : \mathtt{T_r} + \{\mathtt{x} : \mathtt{T}\}}$$

$$(\mathrm{ACCt}_{\mathcal{R}})\qquad \frac{\Gamma \vdash_{\mathcal{R}} r : \mathtt{T_r} \quad \mathtt{T_r}(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash_{\mathcal{R}} r \cdot \mathtt{x} : \mathtt{T}}$$

Figure 4. Type System of $\lambda_{\mathcal{R}}$.

sion e'".The operational semantics of $\lambda_{\mathcal{R}}$ is mostly standard. Evaluation rules and the definition of values are given in Figure 3.

Type System

A monomorphic type system for $\lambda_{\mathcal{R}}$ is given in Figure 4. Types are defined as follows:

$$\begin{array}{lll} \mathit{Type}_{\mathcal{R}} & \mathtt{T} & ::= \iota \mid \mathtt{T} \to \mathtt{T} \mid \mathtt{T_r} \\ \mathit{Record\ Type} & \mathtt{T_r} & \in \mathit{Label} \xrightarrow{\mathit{fin}} \mathit{Type}_{\mathcal{R}} \\ \mathit{Type\ Environment}_{\mathcal{R}} & \Gamma & \in \mathit{Var}_{\mathcal{R}} \xrightarrow{\mathit{fin}} \mathit{Type}_{\mathcal{R}} \end{array}$$

There are base type and function types as usual. A record type is a mapping from field labels to types. Type environments are similar to those for $\lambda_{\mathcal{S}}$.

3. Translation

In this section we present how staged expressions can be represented with record calculus expressions.

We begin with an observation: Boxed expressions are not executed – they remain frozen – until they are `run`. This notion is very similar to closures;; closures are not executed until they are applied to an operand. This observation hints us that boxed expressions can be represented as functions.

The second observation is that when an unboxed expression is replaced with a code template (see rule UNB in the operational semantics), the free variables in the code template may be captured by the surrounding expression. In other words, the surrounding boxed expression provides the code template with an environment that carries the "meaning" of the free variables in the code template. Combining the two observations, we can then represent a boxed expression as a function whose parameter is an environment. Providing an environment to a boxed expression (as in the case of unboxing) is then nothing but a function application where the operator is the boxed expression and the operand is the environment.

The next question is how to represent environments. The answer is trivial: as records. A variable occurrence then becomes a lookup in the current environment (i.e. record), and a binding is an update to the current environment (i.e. record).

Our translation at the type level translates code expression of type $\Box(\Gamma \rhd \mathtt{T})$ into function expression of type $\underline{\Gamma} \to \underline{\mathtt{T}}$, where $\underline{\Gamma}$ is a record type for Γ and $\underline{\mathtt{T}}$ is a translated type for $\mathtt{T}$.

To give a few examples, consider the expression $\mathtt{box}\ x$. It can be represented as $\lambda\rho.\rho\cdot\mathtt{x}$, where the value of $\mathtt{x}$ is being obtained from the environment ρ. The expression

$$\mathtt{box}\ (\mathtt{let}\ \ x = 42\ \mathtt{in}\ \ \mathtt{unbox}\ (\mathtt{box}\ x))$$

can be represented as

$$\lambda\rho'.(\lambda\rho.\rho\cdot\mathtt{x})\{\mathtt{x} = 42\}.$$

Note how the `unbox` expression becomes a function application. As a special case, `run` becomes an application where the argument is the empty environment, because only closed expressions can be executed. For example, $\mathtt{run}\ (\mathtt{box}\ 42)$ becomes $(\lambda\rho.42)\{\}$.

To illustrate how variable capturing is handled, let us now take the following example.

$$\begin{array}{lll} \mathtt{let} & a & = \mathtt{box}\ x \\ & b & = \mathtt{box}\ (\lambda x.\lambda y.(\mathtt{unbox}\ a) + y) \\ \mathtt{in} & \multicolumn{2}{l}{(\mathtt{run}\ b)\ 1\ 1} \end{array}$$

In the example, the value of b will be $\mathtt{box}\ (\lambda x.\lambda y.x+y)$. Note how the variable x, which was free in $\mathtt{box}\ x$, is captured. Continuing the evaluation, $\mathtt{run}\ b$ will reduce to the function $\lambda x.\lambda y.x+y$, resulting in a final value of 2.

Based on the translation described, the example above is translated as below:

$$\begin{array}{lll} \mathtt{let} & a & = \lambda\rho.\rho\cdot\mathtt{x} \\ & b & = \lambda\rho.\lambda x.\lambda y.(a\ (\rho + \{\mathtt{x} = x, \mathtt{y} = y\})) + y \\ \mathtt{in} & \multicolumn{2}{l}{(b\ \{\})\ 1\ 1} \end{array}$$

Both `box` expressions are converted to a function that takes as parameter an environment, ρ. In the first line, the occurrence of x is free. So it is translated to a lookup operation in ρ. The occurrence of y in the second line is not free, hence it is left as it is. The `unbox` expression becomes a function application where the operand is the

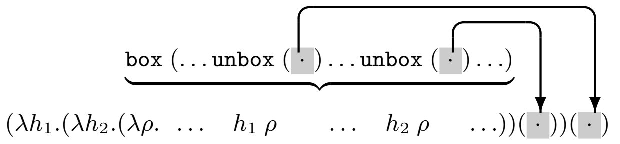

Figure 5. Illustration of the translation of a `box` expression with two `unbox`es.

environment ρ updated with the bindings of x and y. Finally, the run expression is translated to an application to the empty record. When evaluated, the translation reduces to 2, too.

Order of Evaluation

In the staged calculus, `unbox` expressions inside `box` are evaluated to code templates. When translated to record calculus as discussed above, however, the contents of a `box` become guarded under a lambda abstraction and hence are not evaluated. Consider the following example

$$\begin{array}{l} \texttt{box}\ (\texttt{unbox}\ ((\lambda x.x)\ \texttt{box}\ 1)) \\ \xrightarrow{0} \texttt{box}\ (\texttt{unbox}\ (\texttt{box}\ 1)) \\ \xrightarrow{0} \texttt{box}\ 1 \end{array}$$

The translation of $\texttt{box}\ (\texttt{unbox}\ ((\lambda x.x)\ \texttt{box}\ 1))$ would be $\lambda\rho.((\lambda x.x)\ \lambda\rho.1)\rho$, which is already a value and does not evaluate further. So, the order of evaluation (in the call-by-value semantics) is perturbed by the translation. This would incur a serious problem in the presence of expressions with side-effects.

To preserve the order of evaluation, the translation has to move the expression inside `unbox` to the outside of the enclosing `box`, as illustrated in Figure 5. To do this, every `unbox` expression is replaced with a *hole variable* h and a *context* in the form of $(\lambda h.[\cdot])\ \underline{e}$, where $\underline{e}$ is the translation of the unboxed expression, is created so that the inside of the context can be filled in with the translation of the enclosing `box`. Because $\underline{e}$ is at the argument position of a function, the call-by-value semantics of the record calculus then handles the rest and evaluates $\underline{e}$ first. The correct translation of the example above is $(\lambda h.\lambda\rho.h\ \rho)\ \ ((\lambda x.x)\ \lambda\rho.1)$.

Our translation to preserve the order of evaluation is similar to Davies and Pfenning's [14]. They suggested the translation from the implicit modal language Mini-ML$^{\Box}$, which is similar to $\lambda_{\mathcal{S}}$, to the explicit modal language Mini-ML$^{\Box}_{e}$. Their target language is still staged whereas ours is the record language with no staging. Kameyama et.al [24] also developed a similar translation that translates 2-staged programs to System F with tuples. More details on the related work are given in Section 5.

Admin Reductions

Let us examine the evaluation of the expression above in small steps.

$$\begin{array}{l} (\lambda h.\lambda\rho.h\ \rho)\ \ ((\lambda x.x)\ \lambda\rho.1) \\ \xrightarrow{\mathcal{R}} (\lambda h.\lambda\rho.(h\ \rho))\ \ (\lambda\rho.1) \\ \xrightarrow{\mathcal{R}} \lambda\rho.((\lambda\rho.1)\ \rho) \end{array}$$

The final value, $\lambda\rho.((\lambda\rho.1)\ \rho)$, is not directly the translation of $\texttt{box}\ 1$; there is still a reducible term, $(\lambda\rho.1)\ \rho$, inside a lambda. This residual term is seen because of the following fact: In the staged calculus, when an `unbox` expression evaluates to a code template, the code template immediately (i.e. in one step) replaces the `unbox` expression. On the other hand, in the record calculus, the unbox expression becomes an argument to a function, in which the argument is applied to an environment. Passing the argument to the function takes one step of evaluation (i.e. substitution). The application of the argument to the environment still remains, and is, in fact, the residual term that needs to be reduced via further action. This kind of a reduction is called an "admin reduction". Admin reductions simplify the record calculus terms and bring them to a form that is the direct result of a translation. In general, beta-reduction of an application where the operator is a lambda expression and the operand is a record is an admin reduction; this reduction may happen anywhere, including inside lambda abstractions. The example above is admin-reduced as follows, where the admin-reducible term is underlined. Note that the resulting term, $\lambda\rho.1$, is directly the translation of $\texttt{box}\ 1$.

$$\lambda\rho.(\underline{(\lambda\rho.1)\ \rho}) \xrightarrow{\mathcal{A}} \lambda\rho.1$$

There are two kinds of admin reductions. The first is the one explained above. The second is related to variable capture. Recall that when a code template replaces an unbox expression, the free variables are captured. A free variable becomes a lookup expression in the current environment after the translation. Such lookups need to be resolved (after a hole replacement). This is done by the second kind of admin reduction. The following trace belongs to the first example given in this section. Both kinds of admin reductions are used. Admin-reducible terms are again underlined.

$$\begin{array}{l} \texttt{let}\ a\ =\ \lambda\rho.\rho\cdot\texttt{x} \\ \quad\ b\ =\ (\lambda h.\lambda\rho.\lambda x.\lambda y.(h\ (\rho + \{\texttt{x} = x, \texttt{y} = y\})) + y)a \\ \texttt{in}\ \ (b\ \{\})\ 1\ 1 \\ \xrightarrow{\mathcal{R}} \\ \texttt{let}\ b\ =\ (\lambda h.\lambda\rho.\lambda x.\lambda y.(h\ (\rho + \{\texttt{x} = x, \texttt{y} = y\})) + y)(\lambda\rho.\rho\cdot\texttt{x}) \\ \texttt{in}\ \ (b\ \{\})\ 1\ 1 \\ \xrightarrow{\mathcal{R}} \\ \texttt{let}\ b\ =\ \lambda\rho.\lambda x.\lambda y.(\underline{(\lambda\rho.\rho\cdot\texttt{x})\ (\rho + \{\texttt{x} = x, \texttt{y} = y\})}) + y \\ \texttt{in}\ \ (b\ \{\})\ 1\ 1 \\ \xrightarrow{\mathcal{A}} \\ \texttt{let}\ b\ =\ \lambda\rho.\lambda x.\lambda y.(\underline{((\rho + \{\texttt{x} = x, \texttt{y} = y\})\cdot\texttt{x})}) + y \\ \texttt{in}\ \ (b\ \{\})\ 1\ 1 \\ \xrightarrow{\mathcal{A}} \\ \texttt{let}\ b\ =\ \lambda\rho.\lambda x.\lambda y.x + y \\ \texttt{in}\ \ (b\ \{\})\ 1\ 1 \\ \xrightarrow{\mathcal{R}} ((\lambda\rho.\lambda x.\lambda y.x + y)\ \{\})\ 1\ 1 \\ \xrightarrow{\mathcal{R}} (\lambda x.\lambda y.x + y)\ 1\ 1 \\ \xrightarrow{\mathcal{R}} (\lambda y.1 + y)\ 1 \\ \xrightarrow{\mathcal{R}} 1 + 1 \\ \xrightarrow{\mathcal{R}} 2 \end{array}$$

The evaluation of the staged expression is below. Note that any term below translates to a term above.

Definitions

$$\begin{array}{ll} \textit{Environment} & r ::= \{\} \mid \rho \mid r{+}\{\mathtt{x}{=}x\} \\ \textit{Environment Stack} & R ::= \bot \mid R, r \\ & \\ \textit{Context} & \kappa ::= ((\lambda h.[\cdot])\ e) \mid ((\lambda h.\kappa)\ e) \\ \textit{Context Stack} & K ::= \bot \mid K, \kappa \end{array}$$

Environment Lookup

$$r(\mathtt{x}) = \begin{cases} x & \text{if } r = r'{+}\{\mathtt{x}{=}x\} \\ r'(\mathtt{x}) & \text{if } r = r'{+}\{\mathtt{y}{=}\_\} \text{ and } \mathtt{x} \neq \mathtt{y} \\ \rho{\cdot}\mathtt{x} & \text{if } r = \rho \end{cases}$$

Term Translation

(TCON) $$R \vdash i \mapsto (i, \bot)$$

(TVAR) $$R, r \vdash x \mapsto (r(\mathtt{x}), \bot)$$

(TABS) $$\frac{R, r{+}\{\mathtt{x}{=}x\} \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \lambda x.e \mapsto (\lambda x.\underline{e}, K)}$$

(TFIX) $$\frac{R, r{+}\{\mathtt{x}{=}x\}{+}\{\mathtt{f}{=}f\} \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \mathtt{fix}\ fx.e \mapsto (\mathtt{fix}\ fx.\underline{e}, K)}$$

(TAPP) $$\frac{R \vdash e_1 \mapsto (\underline{e_1}, K_1) \quad R \vdash e_2 \mapsto (\underline{e_2}, K_2)}{R \vdash e_1\ e_2 \mapsto (\underline{e_1}\ \underline{e_2}, K_1 \bowtie K_2)}$$

(TBOX) $$\frac{R, \rho \vdash e \mapsto (\underline{e}, (K, \kappa))}{R \vdash \mathtt{box}\ e \mapsto (\kappa[\lambda\rho.\underline{e}], K)}\ \text{new } \rho$$

$$\frac{R, \rho \vdash e \mapsto (\underline{e}, \bot)}{R \vdash \mathtt{box}\ e \mapsto (\lambda\rho.\underline{e}, \bot)}\ \text{new } \rho$$

(TUNB) $$\frac{R \vdash e \mapsto (\underline{e}, K)}{R, r \vdash \mathtt{unbox}\ e \mapsto (h\ r, (K, (\lambda h.[\cdot])\ \underline{e}))}\ \text{new } h$$

(TRUN) $$\frac{R \vdash e \mapsto (\underline{e}, K)}{R \vdash \mathtt{run}\ e \mapsto (\mathtt{let}\ \ h = \underline{e}\ \mathtt{in}\ (h\,\{\}), K)}\ \text{new } h$$

Context Stack Merge Operator

$$\begin{array}{c} \bot \bowtie K = K \\ K \bowtie \bot = K \\ (K_1, \kappa_1) \bowtie (K_2, \kappa_2) = (K_1 \bowtie K_2), (\kappa_1[\kappa_2]) \end{array}$$

Figure 6. Translation from $\lambda_{\mathcal{S}}$ to $\lambda_{\mathcal{R}}$.

$$\begin{array}{l} \mathtt{let}\ a\ =\ \mathtt{box}\ x \\ \quad\ \ b\ =\ \mathtt{box}\ (\lambda x.\lambda y.(\mathtt{unbox}\ a){+}y) \\ \mathtt{in}\ \ (\mathtt{run}\ b)\ 1\ 1 \\ \xrightarrow{0} \\ \mathtt{let}\ b\ =\ \mathtt{box}\ (\lambda x.\lambda y.(\mathtt{unbox}\ (\mathtt{box}\ x)){+}y) \\ \mathtt{in}\ \ (\mathtt{run}\ b)\ 1\ 1 \\ \xrightarrow{0} \\ \mathtt{let}\ b\ =\ \mathtt{box}\ (\lambda x.\lambda y.x{+}y) \\ \mathtt{in}\ \ (\mathtt{run}\ b)\ 1\ 1 \\ \xrightarrow{0} (\mathtt{run}\ (\mathtt{box}\ (\lambda x.\lambda y.x{+}y)))\ 1\ 1 \\ \xrightarrow{0} (\lambda x.\lambda y.x{+}y)\ 1\ 1 \\ \xrightarrow{0} (\lambda y.1{+}y)\ 1 \\ \xrightarrow{0} 1{+}1 \\ \xrightarrow{0} 2 \end{array}$$

The formal definitions of the translation and admin reductions are given in the next section.

3.1 Translation Definition

The translation is presented in Figure 6. A translation judgment has the form $R \vdash e \mapsto (\underline{e}, K)$ with the meaning that "a $\lambda_{\mathcal{S}}$ expression e, under *environment stack* R, translates to the $\lambda_{\mathcal{R}}$ expression $\underline{e}$ and the *context stack* K."

An *environment* is a subset of a record expression that associates fields to variables. It keeps the information of which variables have been bound so far. Each stage has a corresponding environment, held in the environment stack. Hence, the translation of a stage-n expression involves a stack of length $n + 1$. The rightmost (or topmost) environment in the stack corresponds to the current stage.

An expression that binds a variable updates the environment with the new binding. Lambda abstraction and `fix` are such expressions (see rules TABS and TFIX). A `box` expression starts a new environment by putting a fresh environment variable on top of the environment stack. Dually, `unbox` chops off the topmost environment from the stack.

The notion of a *context* was informally discussed in the previous section. A context $((\lambda h.[\cdot])\ \underline{e})$ corresponds to an unboxed expression e where $\underline{e}$ is the translation of e. Contexts are used for putting the unboxed expression outside their enclosing box expressions so that the evaluation order is preserved. The variable that a context binds, that is h, is a fresh variable that replaces the original `unbox` in the translation. Note that there may be multiple `unbox` expressions at a particular stage, e.g. $\mathtt{box}\ (\mathtt{unbox}\ (e_1) + \mathtt{unbox}\ (e_2))$. Therefore, contexts are defined recursively, as in $((\lambda h.\kappa)\ \underline{e})$. This way, a context is able to keep information about multiple `unbox` expressions in a stage, while still preserving their relative order of evaluation. Also note that `unbox` expressions can be nested, e.g. $\mathtt{box}\ (\mathtt{box}\ (\mathtt{unbox}\ (\mathtt{unbox}\ e)))$. The translation, therefore, produces context stacks instead of a single context. Each context in the stack corresponds to a stage. The contexts in a stack are positioned in the following order: The context of the stage that is immediately lower than the current stage is positioned at the rightmost side; stages go lower (i.e. get closer to 0) as we go left. The stage closest to 0 is located at the leftmost side of the stack.

New contexts in the translation are populated by `unbox` expressions (see rule TUNB). A fresh hole variable is also generated as a placeholder for the unboxed expression. The translation of a `box` expression pulls the topmost context from the stack and puts the translated expression inside this context. The translation of expressions with no subexpressions (e.g. variables) results in empty context stacks, since there are no `unbox` contained within the expression. The translation of expressions with single subexpressions (e.g. abstraction) simply threads the context stack that results from the translation of the subexpression. The translation of expressions with more than one subexpression (e.g. application) merges the context stacks resulting from the translation of subexpressions. A context stack merge operation respects the order of appearance, hence serves the preservation of the order of evaluation.

When discussing the translation informally, we converted `run` to a function application, but in the formal definition we translate to a let-expression. The difference is merely syntactic; we want to be able to distinguish translations of `run` from `unbox` so that the inverse translation can properly translate expressions back.

3.2 Semantics Preservation

In this section we formally make the connection between semantics of $\lambda_{\mathcal{S}}$ and $\lambda_{\mathcal{R}}$ through the translation. For complete proofs for lemmas and theorems, we refer the reader to the companion technical report [7].

Recall that a translation yields a pair of an expression and a context stack. This pair can be constructed into a single expression using a context closure operation:

Definition 1. *(Context Closure) Let e be a $\lambda_{\mathcal{R}}$ expression and K be a context stack. The context closure $K(e)$ is defined as follows.*

$$
\begin{tikzcd}
e \arrow[r, "n"] \arrow[d] & e' \arrow[d] \\
\underline{e} & \underline{e}'
\end{tikzcd}
\quad \Longrightarrow \quad
\underline{e} \xrightarrow{\mathcal{R};\mathcal{A}^*} \underline{e}'
$$

Figure 7. Relation between $\lambda_{\mathcal{S}}$ and $\lambda_{\mathcal{R}}$ operational semantics.

$$K(e) = \begin{cases} K'(\kappa[e]) & \textit{if } K = (K', \kappa) \\ e & \textit{if } K = \bot \end{cases}$$

In Section 3 we discussed the need for admin reductions; here we give the formal definition:

Definition 2. *(Admin Reduction) Administrative reduction of an expression is a congruence closure of the following two rules:*

$$\textit{(APP)} \qquad (\lambda\rho.e)\ r \xrightarrow{\mathcal{A}} [\rho \mapsto r]e$$

$$\textit{(ACC)} \qquad \frac{r \neq \rho}{r \cdot \mathtt{x} \xrightarrow{\mathcal{A}} r(\mathtt{x})}$$

The definition of administrative reductions also extends to contexts and context stacks.

Note that an administrative reduction may happen anywhere, even under lambdas. Also note that an admin reduction is "safe" to perform, in the sense that no side-effecting or non-terminating expression is eliminated by an admin reduction. It is also straightforward to check that admin reductions terminate.

Definition 3. *(Admin-normal form) An expression e is said to be in admin-normal form iff there does not exist any e' such that $e \xrightarrow{\mathcal{A}} e'$.*

An important observation is that a translated expression does not contain any admin-reducible terms:

Lemma 1. *Let e be a $\lambda_{\mathcal{S}}$ expression such that $R \vdash e \mapsto (\underline{e}, K)$ for some R. Then, $K(\underline{e})$ is in admin-normal form.*

Proof. By structural induction on e [7]. ☐

Notation 1. *The Kleene closure of admin reductions is denoted as $\xrightarrow{\mathcal{A}^*}$.*

Notation 2. *We use $\xrightarrow{\mathcal{R};\mathcal{A}^*}$ to denote sequential application of one step of eager evaluation followed by exhaustive administrative reductions. Exhaustive admin reductions are those that bring an expression to the admin-normal form.*

Next, we show the relation between the operational semantics of $\lambda_{\mathcal{S}}$ and $\lambda_{\mathcal{R}}$: Given a $\lambda_{\mathcal{S}}$ expression e, we can first translate e, then evaluate it in record language semantics followed by application of admin reductions, and we will have obtained the translation of the expression that e evaluates to in the staged semantics. Furthermore, the admin reductions that we apply are exhaustive; we do not need to worry about oversimplification. This relation is formally stated in Theorem 1 and illustrated in Figure 7.

Two properties are critical to prove the semantic preservation. First the translation preserves the substitution operation.

Lemma 2. *(Substitution Preservation) Assume e_1 is a stage-n $\lambda_{\mathcal{S}}$ expression, e_2 is a stage-0 $\lambda_{\mathcal{S}}$ expression with no free variables. Let $r_0 \dots r_n \vdash e_1 \mapsto (\underline{e_1}, \kappa_p \dots \kappa_1)$ for $p \leq n$ and $\{\} \vdash e_2 \mapsto (\underline{e_2}, \bot)$ where r_0 is such that $r_0(\mathtt{x}) = x$ for some variable x. Then*

- *If $n = 0$ then $r_0 \vdash [x \overset{0}{\mapsto} e_2]e_1 \mapsto ([x \mapsto \underline{e_2}]\underline{e_1}, \bot)$.*
- *If $n > p$ then $r_0 \dots r_n \vdash [x \overset{n}{\mapsto} e_2]e_1 \mapsto (\underline{e_1}, \kappa_p \dots \kappa_1)$.*
- *If $n = p$ then $r_0 \dots r_n \vdash [x \overset{n}{\mapsto} e_2]e_1 \mapsto (\underline{e_1}, (\kappa'_p, \kappa_{p-1} \dots \kappa_1))$ where $\kappa'_p = [x \mapsto \underline{e_2}]\kappa_p$.*

Proof. By structural induction on expression e_1 [7]. ☐

Second the translation preserves the variable-capturing reduction which happens in $\lambda_{\mathcal{S}}$ because of open code.

Lemma 3. *(Variable-Capturing Preservation) Assume e is a stage-n $\lambda_{\mathcal{S}}$ expression and S is a substitution where $S = [\rho \mapsto r]$. Let $r_0 \dots r_n \vdash e \mapsto (\underline{e}, K)$ and $S(r_0 \dots r_n) \vdash e \mapsto (\underline{e}', K')$. Then, $S\underline{e} \xrightarrow{\mathcal{A}^*} \underline{e}'$ and $SK \xrightarrow{\mathcal{A}^*} K'$.*

Proof. By structural induction on e [7]. The substitution operations ($S(r_0 \dots r_n)$, $S\underline{e}$, and SK) are the usual compositional, homomorphic operations. ☐

Finally we give the simulation theorem that shows our translation is semantics-preserving. An illustration of this theorem is given in Figure 7.

Theorem 1. *(Simulation) Let e be a stage-n $\lambda_{\mathcal{S}}$ expression with no free variables such that $e \xrightarrow{n} e'$. Let $R \vdash e \mapsto (\underline{e}, K)$ and $R \vdash e' \mapsto (\underline{e}', K')$. Then $K(\underline{e}) \xrightarrow{\mathcal{R};\mathcal{A}^*} K'(\underline{e}')$.*

Proof. By induction on the evaluation $e \xrightarrow{n} e'$ using Lemma 1, Lemma 2 and Lemma 3. For complete proof, see [7]. ☐

Type Term

$$\iota \mapsto \iota \qquad \frac{\mathtt{T}_1 \mapsto \underline{\mathtt{T}_1} \quad \mathtt{T}_2 \mapsto \underline{\mathtt{T}_2}}{\mathtt{T}_1 \to \mathtt{T}_2 \mapsto \underline{\mathtt{T}_1} \to \underline{\mathtt{T}_2}} \qquad \frac{\Gamma \mapsto \mathtt{T_r} \quad \mathtt{T} \mapsto \underline{\mathtt{T}}}{\Box(\Gamma \rhd \mathtt{T}) \mapsto \mathtt{T_r} \to \underline{\mathtt{T}}}$$

Record Type Term

$$\frac{\Gamma \mapsto \mathtt{T_r} \quad \mathtt{T} \mapsto \underline{\mathtt{T}}}{\Gamma + x : \mathtt{T} \mapsto \mathtt{T_r} + \{\mathtt{x} : \underline{\mathtt{T}}\}} \qquad \varnothing \mapsto \varnothing$$

Figure 8. Type Translation.

Type Translation

A relation between the two languages exists not only between their operational semantics but also between their type systems. The translation preserves the typability of an expression: If a $\lambda_{\mathcal{S}}$ expression is typable in the $\lambda_{\mathcal{S}}$ type system, its translation is typable in the $\lambda_{\mathcal{R}}$ type system. The type translation is a straightforward conversion that converts all the box-types in a $\lambda_{\mathcal{S}}$ type to arrow types. (Figure 8)

Theorem 2. *(Type Correctness) Let e be a stage-0 $\lambda_{\mathcal{S}}$ expression with no free variables such that $\varnothing \vdash_{\mathcal{S}} e : \mathtt{T}$. If $R \vdash e \mapsto (\underline{e}, \bot)$ then $\varnothing \vdash_{\mathcal{R}} \underline{e} : \underline{\mathtt{T}}$.*

For a proof of this theorem, see [1, 2, 37].

3.3 Inverse Translation

We have so far seen how a $\lambda_{\mathcal{S}}$ expression can be translated to a $\lambda_{\mathcal{R}}$ expression and how the two expressions relate. We can also translate a $\lambda_{\mathcal{R}}$ expression back to $\lambda_{\mathcal{S}}$. With such an inverse translation, we can not only translate a $\lambda_{\mathcal{S}}$ expression and evaluate the result using record language semantics as we saw in the previous section, but also translate the evaluation result back to $\lambda_{\mathcal{S}}$ without ever having to evaluate the original $\lambda_{\mathcal{S}}$ expression.

The definition of the inverse translation is in Figure 9. An inverse translation judgment is in the form $H \vdash \underline{e} \mapsto e$ with the meaning that "under the *hole environment* H, the $\lambda_{\mathcal{R}}$ expression $\underline{e}$ translates to the $\lambda_{\mathcal{S}}$ expression e."

Definitions

Hole Environment $\quad H : Var_H \rightarrow Expr_{\mathcal{R}}$

Term Translation

(IVAR) $$H \vdash x \mapsto x$$

(IACC) $$H \vdash \underline{e}\cdot\mathtt{x} \mapsto x$$

(IABS) $$\frac{H \vdash \underline{e} \mapsto e}{H \vdash \lambda x.\underline{e} \mapsto \lambda x.e}$$

(IABS) $$\frac{H \vdash \underline{e} \mapsto e}{H \vdash \mathtt{fix}\ fx.\underline{e} \mapsto \mathtt{fix}\ fx.e}$$

(IAPP) $$\frac{H \vdash \underline{e_1} \mapsto e_1 \quad H \vdash \underline{e_2} \mapsto e_2 \quad \underline{e_1} \neq \lambda h.\underline{e} \quad \underline{e_2} \notin Record_{\mathcal{R}}}{H \vdash \underline{e_1}\ \underline{e_2} \mapsto e_1\ e_2}$$

(ICTX) $$\frac{H \cup \{h : \underline{e'}\} \vdash \underline{e} \mapsto e}{H \vdash ((\lambda h.\underline{e})\ \underline{e'}) \mapsto e}$$

(IBOX) $$\frac{H \vdash \underline{e} \mapsto e}{H \vdash \lambda\rho.\underline{e} \mapsto \mathtt{box}\ e}$$

(IUNB) $$\frac{H \vdash H(h) \mapsto e}{H \vdash h\ r \mapsto \mathtt{unbox}\ e}$$

(IRUN) $$\frac{H \vdash \underline{e} \mapsto e}{H \vdash \mathtt{let}\ \ h = \underline{e}\ \mathtt{in}\ \ (h\ \{\}) \mapsto \mathtt{run}\ e}$$

Figure 9. Inverse Translation from $\lambda_{\mathcal{R}}$ to $\lambda_{\mathcal{S}}$.

A *hole environment* is a function that associates hole variables with expressions. Recall that a forward translation replaces an unbox expression with a hole variable h and moves the unboxed expression outside the enclosing box. A hole environment maps the hole variable to the expression that was moved out so that we can convert the hole variable back to an unbox expression. This is done in the (IUNB) rule. Note that in inverse translation we have a single environment as opposed to having a stack of environments (and stack of contexts) in the forward translation. There are two reasons for this: (1) There is no notion of stages in $\lambda_{\mathcal{R}}$. (2) All the hole variables are freshly generated by the forward translation and they are used only once each in unique locations. Hence, it suffices to use a single function to keep the information about hole variables and associated expressions.

The key points of the inverse translation are the following:

- Record lookup expressions are converted back to variables (rule IACC).
- A lambda abstraction that has a record variable as its parameter is converted to a box expression (rule IBOX).
- A function application where the operator is a hole variable is converted to an unbox expression (rule IUNB).
- A new mapping is added to the hole environment when translating a function application where the operator is a lambda abstraction whose parameter is a hole variable (rule ICTX).

Note that the rules of inverse translation are not ambiguous; each rule matches a unique syntactic category. For instance, even though (IABS) and (IBOX) are both defined for lambda abstractions, in the former, the bound variable is a regular variable and in the latter it is a record variable. These two variables come from disjoint sets and are syntactically differentiable. Similarly, hole variables are syntactically distinguishable. This distinction of variables helps us have an unambiguous coverage of expressions.

$$e \xrightarrow{\ n\ } e' \quad \Longrightarrow \quad \begin{array}{ccc} e & & e' \\ \downarrow & & \uparrow \\ \underline{e} & \xrightarrow{\mathcal{R};\mathcal{A}^*} & \underline{e'} \end{array}$$

Figure 10. Given a $\lambda_{\mathcal{S}}$ expression e, we can evaluate its translation in the $\lambda_{\mathcal{R}}$ semantics and then translate the result back to obtain the result that we get from evaluation of the original expression e.

To make the connection between forward translation and inverse translation, we first define how to interpret context stacks as hole environments.

Definition 4. *(From Contexts to Hole Environments) Let K be a context stack. The operation $\overline{K}$ defines a hole environment in the following way:*

$$\begin{array}{lcl} \overline{K} & = & \begin{cases} \varnothing & \text{if } K = \bot \\ \overline{K'} \cup \overline{\kappa} & \text{if } K = K', \kappa \end{cases} \\ \overline{\kappa} & = & \begin{cases} \{h : e\} & \text{if } \kappa = (\lambda h.[\cdot])\ e \\ \overline{\kappa'} \cup \{h : e\} & \text{if } \kappa = (\lambda h.\kappa')\ e \end{cases} \end{array}$$

The lemma below states that we can translate a $\lambda_{\mathcal{S}}$ expression to $\lambda_{\mathcal{R}}$ and then translate the result back to $\lambda_{\mathcal{S}}$ to obtain the same expression.

Theorem 3. *(Inversion) Let e be a $\lambda_{\mathcal{S}}$ expression and R be an environment stack. If $R \vdash e \mapsto (\underline{e}, K)$, then $H \vdash \underline{e} \mapsto e$ for any H such that $\overline{K} \subseteq H$.*

Proof. By induction on the structure of e [7]. □

Combining Theorem 1 with Theorem 3 gives a stronger result: not only that the evaluation of translated $\lambda_{\mathcal{R}}$-program simulate every reduction step of the original $\lambda_{\mathcal{S}}$-program but also that every intermittent $\lambda_{\mathcal{R}}$-expression occurring in the simulation steps can be projected back to its corresponding $\lambda_{\mathcal{S}}$-expression of the $\lambda_{\mathcal{R}}$-evaluation (Figure 10). The existence of such inversion facilitates our development of the projection step, which is the topic of the next section.

4. Projection

Among our three-step (translate, analysis, and project) approach to analyze multi-staged programs, the first two steps have sound foundations. Since we have proven that the translation is semantic-preserving, statically analyzing translated programs can replace analyzing the original subject programs. The analysis for the translated unstaged programs can be proven correct using the conventional static analysis framework such as abstract interpretation [10, 11].

The last step, projecting the analysis results back in terms of the original staged program, needs a condition for its safety. Arbitrary projections can have images that fail to qualify as static analysis results of the original staged program. For example, staged program run `0 is translated into an application $(\lambda\rho.0)\{\}$, and the binding of the empty record to variable ρ has no counterpart in the original staged program's semantics. Hence, a projection whose image is only such an extra binding effect is clearly not a static analysis result of the original program.

A noticeable point about the safety of projections is that the safety is defined in reference to a static analysis of the original staged program. Checking whether the projection image qualifies to be a static analysis result of the original staged program needs the static analysis definition. This requirement is not a dilemma;

static analysis can be always defined mathematically, though it may not be realizable.

A sufficient condition for projection safety is easy to see once we model static analysis in the abstract interpretation framework [10, 11]. Let e be a multi-staged program and $\underline{e}$ be its translated unstaged version. Let $\llbracket e \rrbracket \in D_S$ and $\llbracket \underline{e} \rrbracket \in D_R$ be their concrete semantics over concrete domains D_S and D_R respectively. Static analyses of e and $\underline{e}$ are computations of abstract (approximate) versions of the concrete semantics. Let $\widehat{\llbracket e \rrbracket} \in \hat{D}_S$ and $\widehat{\llbracket \underline{e} \rrbracket} \in \hat{D}_R$ be the abstract semantics. Each pair of concrete and abstract domains is Galois-connected by an adjoined pair of abstraction (α and $\underline{\alpha}$) and concretization functions (γ and $\underline{\gamma}$). A concrete (resp. abstract) projection π (resp. $\hat{\pi}$) is a monotonic function from D_R to D_S (resp. $\hat{D}_R$ to $\hat{D}_S$). The following diagram summarizes the setting:

$$\begin{array}{ccccc}
e & & \llbracket e \rrbracket \in D_S & \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} & \hat{D}_S \ni \widehat{\llbracket e \rrbracket} \\
\downarrow & & \uparrow \pi & & \uparrow \hat{\pi} \\
\underline{e} & & \llbracket \underline{e} \rrbracket \in D_R & \underset{\underline{\alpha}}{\overset{\underline{\gamma}}{\leftrightarrows}} & \hat{D}_R \ni \widehat{\llbracket \underline{e} \rrbracket}
\end{array}$$

A safety condition for the abstract projection $\hat{\pi}$ is as follows.

Theorem 4. *(Safe Projection) Let e and $\underline{e}$ be a staged program and its translated unstaged version. If $\llbracket e \rrbracket \sqsubseteq \pi \llbracket \underline{e} \rrbracket$ and $\alpha \circ \pi \circ \underline{\gamma} \sqsubseteq \hat{\pi}$ then $\alpha \llbracket e \rrbracket \sqsubseteq \hat{\pi} \widehat{\llbracket \underline{e} \rrbracket}$.*

Proof. By the first condition and the abstraction function α's monotonicity (because of the Galois connection), $\alpha \llbracket e \rrbracket \sqsubseteq (\alpha \circ \pi) \llbracket \underline{e} \rrbracket$, which, by the monotonicity of α and π, and by the correctness of $\widehat{\llbracket \underline{e} \rrbracket}$, is $\sqsubseteq (\alpha \circ \pi \circ \underline{\gamma}) \widehat{\llbracket \underline{e} \rrbracket}$, which, by the second condition, is $\sqsubseteq \hat{\pi} \widehat{\llbracket \underline{e} \rrbracket}$. □

These conditions are not particularly constraining. Concrete projection π that satisfies the first condition $\llbracket e \rrbracket \sqsubseteq \pi \llbracket \underline{e} \rrbracket$ always exists. Such π is the inverse translation function in Section 3.3 composed with an eraser function that first filter out from $\llbracket \underline{e} \rrbracket$, if any, extra things outside $\llbracket e \rrbracket$. Such composition satisfies the condition because (1) $\llbracket \underline{e} \rrbracket$ always includes $\llbracket e \rrbracket$ since the translated program $\underline{e}$ simulates every reduction step of e and (2) by Theorem 3. The second condition is analogous to the usual correctness condition for an abstract operation in the abstract interpretation framework.

Once the above conditions are satisfied, we can concentrate on defining an abstract analysis of $\lambda_{\mathcal{R}}$ programs without considering staged constructs. Analyzing the translated program and applying the abstract projection $\hat{\pi}$ to the analysis result achieves a safe analysis result of the original staged program.

4.1 Example

Consider the following staged program e. As in Section 1 we use Lisp's quasi-quote syntax [35] for staging constructs.

$$\begin{array}{lll}
\texttt{let } x = \texttt{`0} & & \text{(* indexed as } \rho_1 \text{ *)} \\
\quad\; y = \texttt{`(,}x\texttt{ + 2)} & & \text{(* indexed as } \rho_2 \text{ *)} \\
\texttt{in run } y & &
\end{array}$$

The translated version $\underline{e}$, is

$$\begin{array}{l}
\texttt{let } x = \lambda\rho_1\texttt{.0} \\
\quad\; y = (\lambda h.(\lambda\rho_2.(h\ \rho_2)\texttt{+2}))\ x \\
\texttt{in y} \{\}
\end{array}$$

First we consider the three concrete components: concrete semantics $\llbracket e \rrbracket$ and $\llbracket \underline{e} \rrbracket$ and concrete projection π. The concrete semantics of the two programs are collecting semantics: collections of values of expressions and variables.

- $\llbracket e \rrbracket$: Collecting semantics $\llbracket e \rrbracket$ of the staged original has entries such as:

$$\begin{array}{rll}
x & \text{has} & \texttt{`0} \\
y & \text{has} & \texttt{`(,}x\texttt{ + 2)} \text{ where } \texttt{,}x \text{ has } \texttt{`0} \\
(\texttt{run } y) & \text{has} & \texttt{2}
\end{array}$$

- $\llbracket \underline{e} \rrbracket$: Collecting semantics $\llbracket \underline{e} \rrbracket$ of the translated version has entries such as:

$$\begin{array}{rll}
x & \text{has} & \langle \lambda\rho_1.\texttt{0}, \emptyset \rangle \quad \text{(* closure value *)} \\
y & \text{has} & \langle \lambda\rho_2.(h\ \rho_2)\texttt{+2}, \{h \mapsto \langle \lambda\rho_1.\texttt{0}, \emptyset \rangle\} \rangle \\
h & \text{has} & \langle \lambda\rho_1.\texttt{0}, \emptyset \rangle \\
\rho_1 & \text{has} & \{\} \quad \text{(* empty record *)} \\
\rho_2 & \text{has} & \{\} \\
(y\ \{\}) & \text{has} & \texttt{2}
\end{array}$$

- π: Projection π that satisfies $\llbracket e \rrbracket \sqsubseteq \pi \llbracket \underline{e} \rrbracket$ is straightforward: it forgets extra bindings (for h, ρ_1, ρ_2) and projects closure value $\langle \lambda\rho_i, \sigma \rangle$ to code expression i whose `unbox`(comma) expression's code are those projected from the environment σ.

  Projecting the closure values of $\lambda_{\mathcal{R}}$ into code values of $\lambda_{\mathcal{S}}$ is essentially identical to the inverse translation in Section 3.3. That is, π projects closure values as follows:

$$\begin{array}{ccc}
\langle \lambda\rho_1.\texttt{0}, \emptyset \rangle & \text{to} & \texttt{`0} \\
\left.\begin{array}{l} \langle\ \lambda\rho_2.(h\ \rho_2)\texttt{+2}, \\ \quad \{h \mapsto \langle \lambda\rho_1.\texttt{0}, \emptyset \rangle\} \\ \rangle \end{array}\right] & \text{to} & \left[\begin{array}{l} \texttt{`(,}x\texttt{ + 2)} \\ \text{where the } \texttt{,}x \text{ position has} \\ \texttt{`0} \end{array}\right.
\end{array}$$

Now we consider the abstract components: $\widehat{\llbracket e \rrbracket}$, $\widehat{\llbracket \underline{e} \rrbracket}$, and $\hat{\pi}$. Note that the static analysis will compute $\widehat{\llbracket \underline{e} \rrbracket}$ and project its results by $\hat{\pi}$ back in terms of the abstract semantic domain of $\widehat{\llbracket e \rrbracket}$. The abstract semantics $\widehat{\llbracket e \rrbracket}$ of the original staged program is only a mathematical definition that will be referenced in checking the safety of $\hat{\pi}$.

- $\widehat{\llbracket e \rrbracket}$: For the abstract semantics $\widehat{\llbracket e \rrbracket}$ of the original staged program e, suppose we abstract a set of code values into a regular term grammar [18, 9].

  In a regular term grammar, each production's rhs is $f(t, ..., t)$ where the function symbol f is a code expression label ρ_i and each argument term t is either a code expression label or a nonterminal symbol of a grammar. The n-th argument term is for the code to be plugged into the n-th `unbox` expression inside the code expression ρ_i.

  For example, production rule

$$S \to \rho_2(\rho_1)$$

  means the set of code values from code expression ρ_2 whose only hole (`unbox` expression) is plugged by the code value from code expression ρ_1.

- $\widehat{\llbracket \underline{e} \rrbracket}$: Suppose our static analysis $\widehat{\llbracket \underline{e} \rrbracket}$ for the translated unstaged programs is defined in a flow-insensitive 0CFA manner. That is, because the analysis will be ignorant about the environment parts for closures, the best such analysis result for $\underline{e}$ would be:

$$\begin{array}{rllcrll}
x & \text{has} & \lambda\rho_1.\texttt{0} & \qquad & \rho_1 & \text{has} & \{\} \\
y & \text{has} & \lambda\rho_2.(h\ \rho_2)\texttt{+2} & & \rho_2 & \text{has} & \{\} \\
h & \text{has} & \lambda\rho_1.\texttt{0} & & (y\ \{\}) & \text{has} & \texttt{2}
\end{array}$$

- $\hat{\pi}$: Lastly, abstract projection $\hat{\pi}$ cast the above analysis results $\widehat{\llbracket \underline{e} \rrbracket}$ back in terms of regular term grammars of $\widehat{\llbracket e \rrbracket}$. Additionally, it filters out those for the translation-induced extra variables h, ρ_1, and ρ_2.

  Of the many ways to safely project 0CFA-closure values (those corresponding to code) into regular term grammars, a safe yet naive projection $\hat{\pi}$ projects 0CFA-closures as follows:

$$\begin{array}{rcl}
\lambda\rho_1.\texttt{0} & \text{to} & S_1 \to \rho_1 \\
\lambda\rho_2.(h\ \rho_2)\texttt{+2} & \text{to} & S_2 \to \rho_2(S)
\end{array}$$

where the nonterminal S represents all code ($S \to S_1|S_2$). The argument term S in the production rule is for the values of the application expression $(h\ \rho_2)$ that encodes the `unbox`(comma) expression inside code expression `` `(,x + 2) ``.

Another more precise projection projects $\lambda\rho_2.(h\ \rho_2)$+2 differently:

$$\lambda\rho_2.(h\ \rho_2)\texttt{+2} \quad \text{to} \quad S_2 \to \rho_2(\rho_1)$$

where the argument term ρ_1 in the production rule is from the analysis result for the h variable, not blindly the "universe"(S) nonterminal.

Both the two abstract projections $\hat{\pi}$ satisfy the safety condition

$$\alpha \circ \pi \circ \underline{\gamma} \sqsubseteq \hat{\pi}.$$

Let us check the more precise projection case. Note that the concretization image (by $\underline{\gamma}$) of a 0CFA-closure $\lambda\rho.body$ is the set of closure $\langle\lambda\rho.body, \overline{\sigma}\rangle$'s for every possible environment σ for the free variables in $body$. The free variables' values are transitively the concretized images of their abstract values computed by $\hat{[\![\underline{e}]\!]}$. Thus, the image of $\alpha \circ \pi \circ \underline{\gamma}$ for $\lambda\rho_2.(h\ \rho_2)$+2 becomes

$$\begin{array}{lcl} \lambda\rho_2.(h\ \rho_2)\texttt{+2} & \xrightarrow{\underline{\gamma}} & \{\langle\lambda\rho_2.(h\ \rho_2)\texttt{+2}, \{h \mapsto \langle\lambda\rho_1.\texttt{0}, \emptyset\rangle\}\rangle\} \\ & \xrightarrow{\pi} & \texttt{`(,}x\texttt{ + 2)} \text{ where } \texttt{,}x \text{ has } \texttt{`0} \\ & \xrightarrow{\alpha} & S \to \rho_2(\rho_1), \end{array}$$

which is equivalent to the abstract projection $\hat{\pi}$'s image.

5. Related Work

A translation that makes the order of evaluation explicit was previously given by Davies and Pfenning [15]. The translation in Figure 6 follows the same principles. Their translation, however, is not an unstaging one. Recently, a program logic for Mini-$\mathrm{ML}_e^{\Box}$ was presented [3] which precisely captures the operational semantics, yet cannot be realizable as an automatic static analysis.

An unstaging translation was previously discussed by Kameyama et al. [24] but have several limitations for our purpose. Their translation is to System F with tuples, needs type and environment classifier [36] annotations, supports only two stages. Finally, they did not prove the translation's semantics preservation property.

Another idea of translation of staged expressions is given by Chen and Xi [5]. They convert boxed expressions to first-order abstract syntax expressions in a second-order language with recursion. An advantage of this representation is that inverse translation becomes straightforward. Chen and Xi use deBruijn indices to represent program variables inside code templates. This has the problem that a binding at a higher stage may disappear or occur unexpectedly [5]. (Such an example is given by Kim et al. [26, §6.4].)

Our translation is in principle similar to Minamide et al. [31]'s closure conversion where free variables inside lambdas become environment loop operations, though their translation is a type-directed one for conventional unstaged programs.

Our translation is an improvement of [1, 2]: we refined the admin reductions to only two kinds and showed that these two admin reductions suffice to reach an admin-normal form that can be converted back to the corresponding staged expression using an inverse translation. The refined proof of semantic preservation and the existence of the inverse translation is new.

From the perspective of Cousot and Cousot's abstract-interpretation-based program transformation framework [12], our unstaging translation can be seen as being derivable by an abstract interpretation of the staged language λ_S.

Most static analyses for multi-staged programs are string analyses for programs that generate code as strings (a.k.a. heterogeneous meta programs), but they are limited to estimate only the shape, not the semantics, of generated code by using a grammar [8, 30, 6] or the parsing stack [16]. Such string analyses can not analyze the semantics of the generated code string.

Multi-staged static type systems [14, 36, 26, 39] and their inference algorithms are limited forms of staged static analyses. Any extension to estimate other properties than types (*á la* effect type systems [28, 23, 38]) is limited to those that can proceed without analyzing the semantics of the generated code. Existing multi-staged static type systems do not have to analyze the generated code because code-generation expression's type comes with the type of the generated code.

Kamin et al. [25]'s data flow analysis of multi-staged programs [25] combines static and dynamic techniques. Our approach is completely static. Smith et al. [34] presented a static analysis of code templates. Their language is two-staged and code templates are not first-class citizens. Variable bindings do not extend beyond the code templates they are defined in. Our approach does not have this limitation.

6. Conclusion

Static analysis of multi-staged programs is challenging because the basic assumption of conventional static analysis no longer holds: the program text itself is no longer a fixed static entity, but rather a dynamically constructed value.

In this article we have presented a semantic-preserving translation of multi-staged programs into unstaged ones and a static analysis framework based on this translation. Our static analysis approach has three steps: (1) we first apply the unstaging translation; (2) we apply conventional static analysis to the unstaged version; (3) we project the analysis results back in terms of the original staged program. As long as the unstaged static analysis is correct w.r.t. the unstaged semantics, and the projection is safe w.r.t the imaginary staged analysis, a sound static analysis for the original staged programs is obtained. Because directly defining a staged static analysis is difficult, our technique makes it possible to use the knowledge and experience in static analyses of conventional unstaged programs without having to develop staged analyses from scratch.

Our semantics-preserving translation handles staging constructs that have been evolved to be useful in practice (typified in Lisp's quasi-quotation): open code as values, unrestricted operations on references and intentional variable-capturing substitutions. We refer the reader to our companion technical report [7] for the reference cases and complete proofs.

Acknowledgment

We thank Sam Kamin, Elsa Gunter and Peter Sestoft for their very valuable feedback on an earlier version of this paper. We thank Iksoon Kim, Ben Lickly, Cristiano Calcagno, Xavier Rival, and all the members of the ROPAS group for criticism, discussion, correction, and encouragement.

References

[1] Baris Aktemur. *Improving Efficiency and Safety of Program Generation*. PhD thesis, University of Illinois at Urbana-Champaign, 2009.

[2] Baris Aktemur. Type-checking program generators using the record calculus. Under review. (faculty.ozyegin.edu.tr/aktemur), 2009.

[3] Martin Berger and Laurence Tratt. Program logics for homogeneous meta-programming. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, to appear, 2010.

[4] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming.

In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 25–36, London, UK, 2000. Springer-Verlag.

[5] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *ACM International Conference on Functional Programming*, pages 275–286. ACM, August 2003.

[6] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, volume 4729 of *Lecture Notes in Computer Science*, pages 374–388, Sydney, Austrailia, November 2006. Springer-Verlag.

[7] Wontae Choi, Baris Aktemur, and Kwangkeun Yi. Semantics preservation proof of an unstaging translation of lisp-like multi-staged languages. Technical Report ROSAEC-2010-009, ROSAEC Center, Seoul National University, (rosaec.snu.ac.kr/publish/techmemo/ROSAEC-2010-009.pdf), 2010.

[8] Aske Simon Christensen, Anders Mller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the Static Analysis Symposium*, pages 1–18. Springer-Verlag, 2003.

[9] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, April 1999. `http://tata.gforge.inria.fr`.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[11] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[12] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages pp.178–190, January 2002.

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

[14] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 258–270. ACM, 1996.

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

[16] Kyung-Goo Doh, Hyunha Kim, and David Schmidt. Abstract parsing: static analysis of dynamically generated string output using LR-parsing technology. In *Proceeeding of the International Static Analysis Symposium*, 2009. Available from `http://santos.cis.ksu.edu/schmidt/dohsas09.pdf`.

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

[18] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, 1984.

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

[20] Nevin Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, October 1992.

[21] Nevin Heintze. Set based analysis of ML programs. In *Proceedings of the SIGPLAN Conference on Lisp and Functional Programming*, pages 306–317, 1994.

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

[23] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[24] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Closing the stage : from staged code to typed closure. In *Proceedings of The ACM Symposium on Partial Evaluation and Program Manipulations*, pages 147–157, 2008.

[25] Sam Kamin, Baris Aktemur, and Michael Katelman. Staging static analyses for program generation. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 1–10, New York, NY, USA, 2006. ACM.

[26] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 257–269, 2006.

[27] 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.

[28] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–57, 1988.

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

[30] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM.

[31] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283. ACM, 1996.

[32] Aleksandar Nanevski and Frank Pfenning. Staged computation with names and necessity. *Journal of Functional Programming*, 15(6):893–939, 2005.

[33] 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.

[34] Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *Journal of Functional Programming*, 13(3):677–708, 2003.

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

[36] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2003.

[37] Makoto Tatsuta. Translation of multi-staged language. Technical Report NII-2010-003E, National Institute of Informatics, Tokyo, (research.nii.ac.jp/TechReports/), 2010.

[38] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

[39] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.