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 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.

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 execution.

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 (à 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 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.

\[
\begin{align*}
\text{run } x & \\
\text{repeat } x & = ((x + 2) \text{ until } \text{cond}; \text{run } x)
\end{align*}
\]

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 \text{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 ‘\text{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

\['0, (0+2), (0+2+2), \ldots\].\]

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:

\[S \rightarrow 0 | S+2.\]

However, in order to analyze the code run by the ‘\text{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

\[
\begin{align*}
x & := \lambda\rho.0; \\
\text{repeat } x & := (\lambda h. (\lambda \rho. (h \rho) + 2)) x \\
\text{until } \text{cond}; \\
(x \{\})
\end{align*}
\]

The translation works as follows.

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

\[
0 \rightarrow \lambda\rho.0
\]

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

\[
\text{run } 0 \rightarrow (\lambda\rho.0)\{\}
\]

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,

\[
\text{run } x \rightarrow \lambda\rho.\rho.x
\]

• 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:

\[
((x + 2) \rightarrow (\lambda h. (\lambda \rho. (h \rho) + 2)) x
\]

The code value of \(x\) will be plugged into its corresponding hole (the place of ‘\(\cdot\)’). 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 \(\lambda\rho. ((\rho.0) \rho) + 2\). After two iterations, \(\lambda\rho. ((\lambda \rho. (\lambda \rho.0) \rho) + 2) \rho + 2\), 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 OCFP. (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 \(V_x\) or \(V_{\rho}\) for the value of expression \(x\) and variable \(\rho\) respectively. Let us first label some expressions including lambdas:

\[
x := \lambda\rho.0; \\
\text{repeat } x := (\lambda h. (\lambda \rho_2. (h \rho_2) + 2)) x \\
\text{until } \text{cond}; \\
(x \{\})
\]

The analysis will deduce set constraints as follows. For brevity, we write ‘\(\lambda\rho_i\)” omitting the body expression for lambdas (OCFA-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 ‘(\(\lambda h. \ldots \) x’); i.e., \(\lambda h\)’s body expression’s value, which is \(\lambda\rho_2\). Hence

\[V_x \ni \lambda\rho_2.\]
Comparisons

Projection

A projection program can have images that fail to qualify as static analyses. The called functions would be \( V_h \), which has \( \lambda \rho_1 \) and \( \lambda \rho_2 \). Thus,

\[ V_1 \equiv 0 \quad (\text{ } V_h \text{ has } \lambda \rho_1 \text{ and } \lambda \rho_1 \text{’s body’s value is } 0 \text{ } ) \]

\[ V_2 \equiv V_1 + 2. \quad (\text{ } V_h \text{ has } \lambda \rho_2 \text{ and } \lambda \rho_2 \text{’s body’s value is } V_1 + 2 \text{ } ) \]

Similarly, from the application expression \( \langle x \{ {} \} \rangle \),

\[ V_1 \equiv 0 \quad (\text{ } V_x \text{ has } \lambda \rho_1 \text{ and } \lambda \rho_1 \text{’s body’s value is } 0 \text{ } ) \]

\[ V_2 \equiv V_1 + 2. \quad (\text{ } V_x \text{ has } \lambda \rho_2 \text{ and } \lambda \rho_2 \text{’s body’s value is } V_1 + 2 \text{ } ) \]

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:

\[ V_1 \rightarrow 0 \ \mid V_1 + 2 \]

\[ V_2 \rightarrow 0 \ \mid V_1 + 2 \]

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 lambda expressions, some lambdas in the above example analyses’ results correspond to the code expressions in the original staged program. For example, analysis result \( V_0 \) for variable \( h \) has \( \lambda \rho_1 \) and \( \lambda \rho_2 \), whereas respectively corresponds to code expressions ‘0’ and ‘\( \langle x, x + 2 \rangle \). That is, code to be plugged into the place of ‘\( x \)’ and ‘\( x + 2 \)’ can be 0 and, recursively, ‘\( \langle x, x + 2 \rangle \)’.

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, \( \lambda \rho_1 \) 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 Fpenning’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 (à 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 \( \text{run } e \)) 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 \( \lambda_S \) and the target unstaged record language \( \lambda_R \). Section 3 defines and proves semantic-preservation of the unstaging translation from \( \lambda_S \) to \( \lambda_R \). 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 \( \lambda_S \) and the target record language \( \lambda_R \). For each, we present the syntax, operational semantics and the type system.

#### 2.1 Multi-Staged Language \( \lambda_S \)

The language \( \lambda_S \) is a typed, call-by-value \( \lambda \)-calculus with staging annotations. It is based on \( \lambda^{\text{sim}} \) [26], simplified by removing hygienic code composition (i.e. \( \lambda' \)), 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**

**Variable** \( x, y, f \in \text{Var}_S \)

\[
\text{Expr}_S \quad e ::= i \mid x \mid \lambda x.e \mid e \mid \text{fix } f.e \\
\mid \text{box } e \mid \text{unbox } e \mid \text{run } e
\]

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

**Operational Semantics**

\( \lambda_S \) has a small-step, call-by-value, operational semantics. Evaluation rules of the language are in Figure 1. The evaluation \( e \xrightarrow{\text{eval}} e' \) 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

Value

\[ \begin{align*}
V_0 & := i \mid \lambda x.e \mid \text{fix} \, f.x.e \mid \text{box} \, v^1 \\
V_n (n > 0) & := i \mid x \mid \lambda x.e \mid V_{n-1} \mid \text{fix} \, f.x.v^n \\
\end{align*} \]

<table>
<thead>
<tr>
<th>Operational Semantics (n \geq 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(APP) - e_1 \xrightarrow{\text{e_2}} e_3</td>
</tr>
</tbody>
</table>
| \begin{align*}
& e \xrightarrow{\text{n \cdot T \cdot e'}} v \in \text{Value}^n \\
& \langle x.e \rangle \xrightarrow{\text{v}} \text{unbox} \, v \xrightarrow{\text{[x \mapsto v]}} e \\
& (\text{fix} \, f.x.e) \xrightarrow{\text{v}} \text{run} \, (\text{unbox} \, v) \xrightarrow{\text{[x \mapsto v]}} \text{fix} \, f.x.e \\
\end{align*} |

<table>
<thead>
<tr>
<th>(BOX)</th>
</tr>
</thead>
</table>
| \begin{align*}
& e \xrightarrow{\text{n \cdot T \cdot e'}} e' \\
& \text{run} \, e \xrightarrow{\text{n \cdot v}} \text{run} \, e' \\
& \text{unbox} \, e \xrightarrow{\text{n+1 \cdot v}} \text{unbox} \, e' \\
\end{align*} |

<table>
<thead>
<tr>
<th>(RUN)</th>
</tr>
</thead>
</table>
| \begin{align*}
& \text{run} \, e \xrightarrow{\text{n \cdot v}} \text{run} \, e' \\
& \text{unbox} \, e \xrightarrow{\text{n+1 \cdot v}} \text{unbox} \, e' \\
\end{align*} |

<table>
<thead>
<tr>
<th>(ABS)</th>
</tr>
</thead>
</table>
| \begin{align*}
& e \xrightarrow{\text{n \cdot T \cdot e'}} e' \\
& \lambda x.e \xrightarrow{\text{n+1 \cdot T \cdot e'}} \lambda x.e' \\
& e \xrightarrow{\text{n+1 \cdot e'}} \text{fix} \, f.x.e \\
\end{align*} |

<table>
<thead>
<tr>
<th>(FIX)</th>
</tr>
</thead>
</table>
| \begin{align*}
& \text{fix} \, f.x.e \xrightarrow{\text{n+1 \cdot e'}} \text{fix} \, f.x.e' \\
\end{align*} |

Figure 1. Operational Semantics of \( \lambda_S \).

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

\( \lambda_S \) 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 \( \lambda_S \). A polymorphic type system is also available [26]. Types in \( \lambda_S \) are defined as below.

<table>
<thead>
<tr>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T ) := ( i \mid T \rightarrow T \mid \square (\Gamma \triangleright T) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type Environments ( \Gamma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \in \text{Var}_S \uplus \text{Ty}_S )</td>
</tr>
</tbody>
</table>

We use \( T \) to denote type terms, \( i \) for base types, \( T \rightarrow T \) for function types, \( \square (\Gamma \triangleright T) \) for code template types, \( \Gamma \) for type environments. A code template is given a box-type \( \square (\Gamma \triangleright T) \) with the meaning that “the code template will evaluate to a value of type \( T \) if put in a context that provides the environment \( \Gamma \).” The type environment \( \Gamma \) in the box-type \( \square (\Gamma \triangleright T) \) contains the types of the unbound variables in the code template.

A type environment \( \Gamma \) is a mapping from variables to types, \( \Gamma \left[ x : T \right] \) is a function update operation that defines a function as follows: \( (\Gamma \left[ x : T \right]) (x) = T \) and \( (\Gamma \left[ y : T \right]) (x) = \Gamma (x) \) if \( x \neq y \). A typing judgment has the form \( \Gamma \vdash x : T \) with the meaning that “a stage-n expression \( e \), under type environments \( \Gamma_0 \ldots \Gamma_n \), has type \( T \).” \( \Gamma_0 \ldots \Gamma_n \) is a sequence of type environments. Each type environment corresponds to a stage where \( \Gamma_n \) 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].

2.2 The Record Calculus \( \lambda_R \)

The language \( \lambda_R \) is a \( \lambda \)-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 \( \lambda_S \), we include let-bindings in \( \lambda_R \). This is to be able to syntactically distinguish several \( \lambda_R \) expressions during inverse translation (Section 3.3). The language is still monomorphic.

Syntax

<table>
<thead>
<tr>
<th>Variable ( \rho )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \in \text{Var}_P ) (record variables)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type ( h )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \in \text{Var}_H ) (hole variables)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Labels ( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \in \text{Label} = { x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expressions ( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e \ ::= i \mid w \mid \lambda \omega.e \mid e \mid e \mid \text{fix} , f.x.e \mid r \mid r \mid \text{let} \ w = e \in e )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Values ( v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v \ ::= i \mid \lambda \omega.e \mid \text{fix} , f.x.e \mid v_r )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record Value ( v_r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_r \ ::= { } \mid v_r \mid { x = v } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Records ( r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r \ ::= { } \mid \rho \mid r \mid { x = x } \mid r \mid { x = v } )</td>
</tr>
</tbody>
</table>

The record language \( \lambda_R \) has constants (\( i \)), variables (\( x \)), lambda abstractions, applications, a fixpoint operator \( \text{fix} \), and let-expressions. As for the record operations there are empty records (\( \{ \} \)), record variables (\( \rho \)), and the record update operations \( r \mid \{ x = v \} \). Field names (or labels) in records, we use variables written in teletype font.

We separate variables into three disjoint sets: ordinary variables \( \text{Var}_X \) (which are the same as variables of \( \lambda_S \)), record variables \( \text{Var}_P \), and hole variables \( \text{Var}_H \). 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

\( \lambda_R \) has a small-step, call-by-value operational semantics. The evaluation \( e \xrightarrow{\text{run}} e' \) means that “the expression evaluates to expres-
Operational Semantics

\[(\text{APP}_R) \quad e_1 \xrightarrow{R} e_1' \quad e_2 \xrightarrow{R} e_2' \quad e \xrightarrow{R} e' \quad v \xrightarrow{R} v' \quad (\lambda w. e) v \xrightarrow{R} [w \mapsto v'] e \quad (f \ x. e) v \xrightarrow{R} [x \mapsto v][f \mapsto f \ x. e] \]

(LET_R) \[\text{let } w = e_1 \text{ in } e_2 \xrightarrow{R} \text{let } w = e'_1 \text{ in } e_2 \]

(ACC_R) \[v_r(x) = \begin{cases} v & \text{if } v_r = v'_r + \{x = v\} \\ v'_r(x) & \text{if } v_r = v'_r + \{y = \_\} \text{ and } x \neq y \end{cases} \]

Figure 3. Operational Semantics of \(\lambda_R\).

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 \(\lambda_S\).

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 to 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 \(\Box (G \to T)\) into function expression of type \(\Gamma \to \overline{T}\), where \(\overline{T}\) is a record type for \(G\) and \(\Gamma\) is a translated type for \(G\).

To give a few examples, consider the expression \(\text{let } y = \text{unbox}(\text{box } x)\). It can be represented as \(\lambda \rho. \rho \ x\), where the value of \(x\) is being obtained from the environment \(\rho\). The expression

\[
\text{box (let } x = 42 \text{ in unbox (box } x))
\]

can be represented as

\[
\lambda \rho'.(\lambda \rho. \rho \ x)(\{x = 42\}).
\]

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

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

\[
\begin{aligned}
\text{let } a = \text{box } x \\
\text{b = box } (\lambda x. \lambda y. (\text{unbox } a + y)) \\
\text{in (run b) 1 1}
\end{aligned}
\]

In the example, the value of \(b\) will be \(\text{box } (\lambda x. \lambda y. x + y)\). Note how the variable \(x\), which was free in box \(x\), is captured. Continuing the evaluation, \(\text{run b}\) 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:

\[
\begin{aligned}
\text{let } a = \lambda \rho. \rho \ x \\
b = \lambda \rho. \lambda x. \lambda y. (a (\rho + \{x = x, y = y\}) + y) \\
\text{in (\{\}) 1 1}
\end{aligned}
\]

Both box expressions are converted to a function that takes as parameter an environment, \(\rho\). In the first line, the occurrence of \(x\) is free. So it is translated to a lookup operation in \(\rho\). 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
environment $\rho$ 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 fol-
lowing example

$$\text{box (unbox ((}(\lambda x.x)\text{ box }1))}$$
$$\xrightarrow{\beta} \text{box (unbox (box }1))$$

The translation of box (unbox (($\lambda x.x$) box 1)) would be
$\lambda \rho.(\lambda x.x)\lambda \rho.1$, 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
$(\lambda h.[])$, where $\epsilon$ 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 $\epsilon$ is at the argument
position of a function, the call-by-value semantics of the record
calculus then handles the rest and evaluates $\epsilon$ first. The correct
translation of the example above is $(\lambda h.(\lambda x.x)\lambda \rho.1)$
($\lambda x.x$ $\lambda \rho.1$).

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$^{\text{IL}}$, which is similar to $\lambda S$,
to the explicit modal language Mini-ML$^{\text{EL}}$. 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 trans-
lates 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.

$$(\lambda h.(\lambda x.x)\lambda \rho.1)$$
$$\xrightarrow{\beta} (\lambda h.(\lambda x.x)\lambda \rho.1)$$
$$\xrightarrow{\beta} \lambda \rho.((\lambda \rho.1)\rho)$$

The final value, $\lambda \rho.((\lambda \rho.1)\rho)$, is not directly the translation of
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 any-
where, including inside lambda abstractions. The example above is
admin-reduced as follows, where the admin-reducible term is under-
lined. Note that the resulting term, $\lambda \rho.1$, is directly the transla-
tion of box 1.

$$\lambda \rho.((\lambda \rho.1)\rho) \xrightarrow{\Delta} \lambda \rho.1$$

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.

$$\begin{align*}
\text{let } a &= \lambda \rho.\rho x \\
\text{let } b &= (\lambda h.\lambda \rho.\lambda x.\lambda y.(\rho + [x = x, y = y])) + y) a \\
\text{in } (\{\}) \ 1 1 \\
\text{let } b &= (\lambda h.\lambda \rho.\lambda x.\lambda y.(\rho + [x = x, y = y])) + y)(\lambda \rho.\rho x) \\
\text{in } (\{\}) \ 1 1 \\
\text{let } b &= \lambda \rho.\lambda x.\lambda y.((\lambda \rho.\rho x) (+ [x = x, y = y])) + y \\
\text{in } (\{\}) \ 1 1 \\
\text{let } b &= \lambda \rho.\lambda x.\lambda y.((\lambda \rho.\rho x) (+ [x = x, y = y])) x) + y \\
\text{in } (\{\}) \ 1 1 \\
\text{let } b &= \lambda \rho.\lambda x.\lambda y.x + y \\
\text{in } (\{\}) \ 1 1 \\
\text{let } b &= ((\lambda \rho.\lambda x.\lambda y.x + y) \{\}) \ 1 1 \\
\text{let } b &= (\lambda \rho.\lambda x.\lambda y.x + y) \{\}) \ 1 1 \\
\text{let } b &= \lambda \rho.\lambda x.\lambda y.x + y \\
\text{in } (\{\}) \ 1 1 \\
\text{let } b &= \lambda \rho.\lambda x.\lambda y.x + y \\
\text{in } (\{\}) \ 1 1 \\
\text{let } b &= \lambda \rho.\lambda x.\lambda y.x + y \\
\text{in } (\{\}) \ 1 1 \\
\end{align*}$$

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

Environment  \( r ::= \emptyset \mid \rho \mid r + \{x = e\} \)

Environment Stack  \( R ::= \emptyset \mid R, r \)

Context  \( n ::= ((\lambda h, []), e) \mid ((\lambda h, \kappa), e) \)

Context Stack  \( K ::= \emptyset \mid K, \kappa \)

Environment Lookup  \( r(x) = \begin{cases} r'(x) & \text{if } r = r' + \{x = e\} \\ \rho & \text{if } r = \rho \end{cases} \)

Term Translation

(TCON)  \( R \vdash i \mapsto (i, \perp) \)

(TVAR)  \( R, r + \{x = e\} \vdash e \mapsto (e, \perp) \)

(TABS)  \( R, r + \lambda x.e \vdash e \mapsto (\lambda x. e, K) \)

(TFIX)  \( R, r + \fix f x.e \vdash e \mapsto (\fix f x.e, K) \)

(TAPP)  \( R \vdash e_1 \mapsto (e_1, K_1) \)

(TBOX)  \( R, \rho + e \vdash (\xi, (K, \kappa)) \vdash e \mapsto (\xi, (K, \kappa), K) \)

(TUNB)  \( R, r + \unbox e \vdash (h, r, (\lambda h, [\cdot]), \perp) \)

(TRUN)  \( R \vdash e \mapsto (\lambda h. e, K) \)

Context Stack Merge Operator

\( \perp \bowtie K = K \)

\( (K_1, \kappa_1) \bowtie K = (K_1 \bowtie K, \kappa_1 \bowtie \kappa) \)

Figure 6. Translation from \( \lambda_S \) to \( \lambda_R \).

3.1 Translation Definition

The translation is presented in Figure 6. A translation judgment has the form \( R \vdash e \mapsto (\xi, K) \) with the meaning “a \( \lambda_S \) expression \( e \), under environment stack \( R \), translates to the \( \lambda_R \) expression \( \xi \) 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 \( ((\lambda h, [\cdot]), \xi) \) corresponds to an unboxed expression \( e \), where \( \xi \) 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 unboxed expressions at a particular stage, e.g. \( \text{box } (\text{unbox } (e_1) + \text{unbox } (e_2)) \). Therefore, contexts are defined recursively, as in \( ((\lambda h, \kappa), \xi) \). This way, a context is able to keep information about multiple unboxed expressions in a stage, while still preserving their relative order of evaluation. Also note that unboxed expressions can be nested, e.g. \( \text{box } (\text{box } (\text{unbox } e)) \). 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 unboxed 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 \( \text{run} \) 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 \( \text{run} \) 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 \( \lambda_S \) and \( \lambda_R \) 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 \( \lambda_R \) expression and \( K \) be a context stack. The context closure \( K(e) \) is defined as follows.

\( \vdash \text{let } a = \text{box } x \)
\( \vdash \text{let } b = \text{box } (\lambda x. \lambda y.(\text{unbox } a)+y) \)
\( \vdash \text{let } b = \text{box } (\lambda x. \lambda y.x+y) \)
\( \vdash \text{run } (\text{run } (\text{run } (\text{run } (\text{run } (\text{run } (\text{run } 1) 1) 1) 1) 1) 1) 1 \)
\( \vdash 1+1 \)
\( \vdash 2 \)

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

\[
\begin{align*}
\text{(APP)} & \quad (\lambda \rho . e) \xrightarrow{\mathcal{A}} \rho \rightarrow [\rho \mapsto e] \\
\text{(ACC)} & \quad r \neq \rho \xrightarrow{\mathcal{A}} r(x) \quad r \neq \rho
\end{align*}
\]

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 if 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 \( \lambda_S \) expression such that \( R \vdash e \rightarrow (e, K) \) for some \( R \). Then, \( K(e) \) is in admin-normal form.

**Proof.** By structural induction on \( e \) [7].

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

**Notation 2.** We use \( \xrightarrow{\mathcal{A}^*} \) 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 \( \lambda_S \) and \( \lambda_R \): Given a \( \lambda_S \) 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-\( n \) \( \lambda_S \) expression, \( e_2 \) is a stage-\( 0 \) \( \lambda_S \) expression with no free variables. Let \( r_0 \ldots r_n \vdash e_1 \rightarrow (e_1, \kappa) \) for \( p \leq n \) and \( \{ \} \vdash e_2 \rightarrow (e_2, \bot) \) where \( r_0 \) is such that \( r_0(x) = x \) for some variable \( x \). Then

- If \( n = 0 \) then \( r_0 \vdash [x \mapsto e_2] e_1 \rightarrow (e_1, \kappa_p \ldots \kappa_1) \).
- If \( n > p \) then \( r_0 \ldots r_n \vdash [x \mapsto e_2] e_1 \rightarrow (e_1, \kappa_p \ldots \kappa_1) \).

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-\( n \) \( \lambda_S \) expression with no free variables such that \( e \xrightarrow{\mathcal{A}} e' \). Let \( R \vdash e \rightarrow (e, K) \) and \( R \vdash e' \rightarrow (e', K') \). Then \( K(e) \xrightarrow{\mathcal{A}^*} K'(e') \).

**Proof.** By structural induction on \( e \) [7]. The substitution operations \( (S[r_0 \ldots r_n], S_S, S_K) \) are the usual compositional, homomorphic operations.

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 \( \lambda_S \) expression is typable in the \( \lambda_S \) type system, its translation is typable in the \( \lambda_R \) type system. The type translation is a straightforward conversion that converts all the box-types in a \( \lambda_S \) type to arrow types. (Figure 8)

**Theorem 2.** (Type Correctness) Let \( e \) be a stage-\( 0 \) \( \lambda_S \) expression with no free variables such that \( \emptyset \vdash e : T \). If \( R \vdash e \rightarrow (e, \bot) \) then \( \emptyset \vdash \emptyset : \mathcal{T} \).

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

### 3.3 Inverse Translation

We have so far seen how a \( \lambda_S \) expression can be translated to a \( \lambda_R \) expression and how the two expressions relate. We can also translate a \( \lambda_R \) expression back to \( \lambda_S \). With such an inverse translation, we can not only translate a \( \lambda_S \) expression and evaluate the result using record language semantics as we saw in the previous section, but also translate the evaluation result back to \( \lambda_S \) without ever having to evaluate the original \( \lambda_S \) expression. The definition of the inverse translation is in Figure 9. An inverse translation judgment is in the form \( H \vdash e \rightarrow e \) with the meaning that “under the hole environment \( H \), the \( \lambda_R \) expression \( e \) translates to the \( \lambda_S \) expression \( e \).”
Definitions

Hole Environment  $H : Var_H \rightarrow Expr_R$

Term Translation

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

(IACC)  $H \vdash e \cdot x \rightarrow x$

(IABS)  $H \vdash \epsilon \rightarrow e$

(ABSI)  $H \vdash \epsilon \rightarrow e$

(IAPP)  $H \vdash e_1 \mapsto e_1 \vdash e_2 \mapsto e_2 \ e_1 \neq \lambda \ h e \ e_2 \not\in \text{Record}_R$

(CTX)  $H \vdash \{(h : e')\} \rightarrow e$

(IBOX)  $H \vdash \lambda p e \rightarrow box e$

(IUNB)  $H \vdash H(h) \rightarrow e$

(IRUN)  $H \vdash \mathsf{let} \ h = \epsilon \mathsf{in} \ (\{h\}) \mapsto \mathsf{run} e$

Figure 9. Inverse Translation from $\lambda_R$ to $\lambda_S$.

A hole environment is a function that associates hole variables with expressions. Recall that a forward translation replaces an unboxed expression with a 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 unboxed 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 $\lambda_R$. (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 unboxed 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.

Figure 10. Given a $\lambda_S$ expression $e$, we can evaluate its translation in the $\lambda_R$ 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 $K$ defines a hole environment in the following way:

$K = \begin{cases} \emptyset & \text{if } K = \bot \\ K \cup \pi & \text{if } K = K', \kappa \end{cases}$

$\pi = \begin{cases} \{h : e\} & \text{if } \kappa = (\lambda h.[]\{\}) e \\ \{h \in \{\}\} & \text{if } \kappa = (\lambda h.\{\}) e \end{cases}$

The lemma below states that we can translate a $\lambda_S$ expression into $\lambda_R$ and then translate the result back to obtain the same expression.

Theorem 3. (Inversion) Let $e$ be a $\lambda_S$ expression and $R$ be an environment stack. If $R \vdash e \rightarrow (\epsilon, K)$, then $H \vdash \epsilon \rightarrow e$ for any $H$ such that $K \subseteq H$.

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 $\lambda_R$-program simulate every reduction step of the original $\lambda_S$-program but also that every intermittent $\lambda_R$-expression occurring in the simulation steps can be projected back to its corresponding $\lambda_S$-expression of the $\lambda_R$-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 $\cdot 0$ is translated into an application $(\lambda p.0\{\})$, and the binding of the empty record to variable $p$ 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 \( \bar{e} \) be its translated unstaged version. Let \( [e] \in D_S \) and \( [\bar{e}] \in D_R \) be their concrete semantics over concrete domains \( D_S \) and \( D_R \) respectively. Static analyses of \( e \) and \( \bar{e} \) are computations of abstract (approximate) versions of the concrete semantics. Let \( [e] \in D_S \) and \( [\bar{e}] \in D_R \) be the abstract semantics. Each pair of concrete and abstract domains is Galois-connected by an adjoined pair of abstraction (\( \alpha \) and \( \alpha \bar{\alpha} \)) and concretization functions (\( \gamma \) and \( \gamma \)). A concrete (resp. abstract) projection \( \pi \) (resp. \( \bar{\pi} \)) is a monotonic function from \( D_R \) to \( D_S \) (resp. \( D_R \) to \( D_S \)). The following diagram summarizes the setting:

\[
\begin{array}{c|c}
\text{e} & [e] \\
\hline
\pi & \gamma \\
\bar{\pi} & \gamma \\
\end{array}
\]

A safety condition for the abstract projection \( \bar{\pi} \) is as follows.

**Theorem 4.** (Safe Projection) Let \( e \) and \( \bar{e} \) be a staged program and its translated unstaged version. If \( [e] \sqsubseteq \pi [\bar{e}] \) and \( \alpha \circ \pi \circ \gamma \subseteq \bar{\pi} \) then \( \alpha [e] \sqsubseteq \pi [\bar{e}] \).

**Proof.** By the first condition and the abstraction function \( \alpha \)'s monotonicity (because of the Galois connection), \( \alpha [e] \subseteq \alpha \circ \pi \circ \gamma [\bar{e}] \), which, by the monotonicity of \( \alpha \) and \( \pi \), and by the correctness of \( [e] \), is \( \alpha \circ \pi \circ \gamma [\bar{e}] \subseteq \bar{\pi} [\bar{e}] \), which, by the second condition, is \( \subseteq \bar{\pi} [\bar{e}] \).

These conditions are not particularly constraining. Concrete projection \( \pi \) that satisfies the first condition \( [e] \subseteq \pi [\bar{e}] \) always exists. Such \( \pi \) is the inverse translation function in Section 3.3 composed with an eraser function that first filter out from \( [\bar{e}] \), if any, extra things outside \( [e] \). Such composition satisfies the condition because (1) \( [\bar{e}] \) always includes \( [e] \) since the translated program \( \bar{e} \) 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 \( \lambda \) programs without considering staged constructs. Analyzing the translated program and applying the abstract projection \( \bar{\pi} \) 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.

\[
\begin{align*}
\text{let } x & = '0 \\
\text{y} & = '(', (x + 2) \text{ where } x \text{ has } '0 \\
\text{run } y
\end{align*}
\]

The translated version \( \bar{e} \) is

\[
\begin{align*}
\text{let } x & = \lambda p_1.0 \\
y & = (\lambda h, (\lambda p_2, (h p_2)+2) \text{ x} \\
\text{in } y \{\}
\end{align*}
\]

First we consider the three concrete components: concrete semantics \([e]\) and \([\bar{e}]\) and concrete projection \(\pi\). The concrete semantics of the two programs are collecting semantics: collections of values of expressions and variables.

- \([e]\): Collecting semantics \([e]\) of the staged original has entries such as:
  \[
  \begin{align*}
x & \text{ has } '0 \\
y & \text{ has } '(',(x + 2) \text{ where } x \text{ has } '0 \\
\text{run } y & \text{ has } 2
\end{align*}
\]

- \([\bar{e}]\): Collecting semantics \([\bar{e}]\) of the translated version has entries such as:
  \[
  \begin{align*}
x & \text{ has } (\lambda p_1,0,0) \text{ (* closure value *)} \\
y & \text{ has } (\lambda p_1, (h p_2)+2, (h \mapsto (\lambda p_1,0,0))) \\
\text{h} & \text{ has } (\lambda p_1,0,0) \\
\text{rho_1} & \text{ has } \{\} \text{ (* empty record *)} \\
\text{rho_2} & \text{ has } \{\} \\
\text{y} \{\} & \text{ has } 2
\end{align*}
\]

- \(\pi\): Projection \(\pi\) that satisfies \([e] \subseteq \pi [\bar{e}]\) is straightforward: it forgets extra bindings (for \( h, \rho_1, \rho_2 \)) and projects closure value \((\lambda \rho_1, \sigma)\) to code expression \(i\) whose unbox(comma) expression's code are those projected from the environment \(\sigma\).

**Projection**

The closure values of \(\lambda \bar{e}\) into code values of \(\lambda \bar{e}\) is essentially identical to the inverse translation in Section 3.3. That is, \(\pi\) projects closure values as follows:

\[
\begin{align*}
(\lambda p_1,0,0) & \text{ to } '0 \\
(h \mapsto (\lambda p_1,0,0)) & \text{ to } '(',(x + 2) \\
\text{where the } x \text{ position has } '0
\end{align*}
\]

Now we consider the abstract components: \([e]\), \([\bar{e}]\), and \(\tilde{\pi}\). Note that the static analysis will compute \([\bar{e}]\) and project its results by \(\tilde{\pi}\) back in terms of the abstract semantic domain of \([e]\). The abstract semantics \([e]\) of the original staged program is only a mathematical definition that will be referenced in checking the safety of \(\tilde{\pi}\).

- \([e]\): For the abstract semantics \([e]\) 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 \(\rho_1\) and each argument term \(t\) is either a code expression label or a non-terminal 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 \(\rho_2\).

For example, production rule

\[
S \rightarrow \rho_2(\rho_1)
\]

means the set of code values from code expression \(\rho_2\) whose only hole (unbox expression) is plugged by the code value from code expression \(\rho_1\).

- \([\bar{e}]\): Suppose our static analysis \([\bar{e}]\) 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 \(\bar{e}\) would be:

\[
\begin{align*}
x & \text{ has } \lambda p_1,0 \\
y & \text{ has } \lambda p_2,(h p_2)+2 \\
\text{h} & \text{ has } \lambda p_1,0 \\
\text{rho_1} & \text{ has } \{\} \\
\text{rho_2} & \text{ has } \{\} \\
\text{y} \{\} & \text{ has } 2
\end{align*}
\]

- \(\tilde{\pi}\): Last, abstract projection \(\tilde{\pi}\) cast the above analysis results \([\bar{e}]\) back in terms of regular term grammars of \([e]\). Additionally, it filters out those for the translation-induced extra variables \(h, \rho_1\), and \(\rho_2\).

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

\[
\begin{align*}
\lambda p_1,0 & \text{ to } S_1 \rightarrow \rho_1 \\
\lambda p_2,(h p_2)+2 & \text{ to } S_2 \rightarrow \rho_2(S)
\end{align*}
\]
For the nonterminal $S$ represents all code $(S \rightarrow S_1 | S_2)$. 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 $\langle \cdot, x + 2 \rangle$.

Another more precise projection projects $\lambda p_2, (h \rho_2) \ast 2$ differently:

$$\lambda p_2, (h \rho_2) \ast 2 \rightarrow S_2 \rightarrow \rho_2 (p_1)$$

where the argument term $p_1$ in the production rule is from the analysis result for the $h$ variable, not blindly the "universe" $(S)$ nonterminal.

Both the two abstract projections $\pi$ satisfy the safety condition

$$\alpha \circ \pi \circ \gamma \subseteq \pi.$$ 

Let us check the more precise projection case. Note that the concretization image (by $\gamma$) of a 0CFA-closure $\lambda p. body$ is the set of closure $(\langle \lambda p. body, \sigma \rangle)'s$ for every possible environment $\sigma$ for the free variables in $body$. The free variables' values are transitive the concretized images of their abstract values computed by $\gamma$. Thus, the image of $\alpha \circ \pi \circ \gamma$ for $\lambda p_2, (h \rho_2) \ast 2$ becomes

$$\lambda p_2, (h \rho_2) \ast 2 \rightarrow S \rightarrow \rho_2 (p_1).$$

which is equivalent to the abstract projection $\pi$'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-ML [17] 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 classes, and their translation is 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 (a 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 the 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 Ilsoom Kim, Ben Lickly, Cristiano Calcagno, Xavier Rival, and all the members of the ROAPAS group for criticism, discussion, correction, and encouragement.

References


