Unstaging Translation of Cross-Stage Persistent Multi-Staged Programs into Context Calculus

Joonwon Choi
Seoul National University
jwchoi@ropas.snu.ac.kr

Abstract. We present a semantic-preserving unstaging translation of cross-stage persistent multi-staged programs into context calculus. This unstaging translation enables static analysis by 1) unstaging the source program, 2) analyzing the unstaged program using the conventional static analysis techniques, and 3) projecting the analysis result back to the source language. Static analysis of multi-staged programs is challenging. The basic assumptions of conventional static analyses are no longer guaranteed, since a program text is a first-class citizen in multi-staged programs. We focus on cross-stage persistent multi-staged programs. Unlike Lisp-like multi-staged programs, cross-stage persistent multi-staged programs allow variables of any stage to be used in all future stages (cross-stage persistence) and do not allow intentional variable-capturing substitution. We find that cross-stage persistent multi-staged programs are naturally unstaged to the context calculus.

1 Introduction

Multi-staged programming is a general principle for code-generation systems, such as macro expansion [21, 9], partial evaluation [14, 6], program manipulation [1], and runtime code generation [8, 19, 17]. Lisp’s quasi-quotation [21, 9], MetaOCaml [2] and TemplateHaskell [20] support multi-staged features.

Static analysis of multi-staged programs is challenging. In multi-staged programs, a code template is a first-class citizen and the program code is no longer statically fixed. Code templates are freely passed like other values, composed of other code templates, executed when appropriate, and substituted in other code templates.

Contributions

– We present an unstaging translation and prove that it preserves the small-step operational semantics by step-by-step simulation. We also present an inverse translation. The target language, the context calculus [10], is easier to design a static analysis of than the source language.
– Unstaging translation enables static analysis of cross-stage persistent (henceforth, CSP) multi-staged programs by 1) unstaging the source program, 2) analyzing the unstaged program using conventional static analysis techniques, and 3) projecting the analysis result back to the source language.
– Our translation supports all fundamental CSP multi-staged features: code substitution, code execution, and cross-stage persistence. This paper omits fixpoint lambda abstractions and references, as their extensions are rather straightforward.

Notation 1 In the examples throughout this paper, we write box e and unbox e for <e> and ~e in MetaML (CSP), and ’e and ,e in Lisp’s quasi-quotation system, respectively.

1.1 Difference in Cross-Stage Persistent and Lisp-Like Multi-Staged Languages

Main difference between CSP and Lisp-like multi-staged language is variable scoping. A variable of CSP multi-staged language is used at all future stages [23], while a variable of Lisp is used only at the same stage. For example,

$$(\lambda x. \text{box } x) 0 \xrightarrow{\text{CSP}} \text{box } 0 \quad (\lambda x. \text{box } x) 0 \xrightarrow{\text{Lisp}} \text{box } x$$
In CSP multi-staged language (on the left) \( x \) in \( \text{box} \) is bound to \( \lambda x \), but in Lisp-like multi-staged language (on the right) \( x \) in \( \text{box} \) is not bound to \( \lambda x \). CSP multi-staged language supports only capture-avoiding substitution. On the other hand, Lisp supports both variable-capturing and capture-avoiding substitution. For example,

\[
(\lambda y. (\text{box} (\lambda x. (\text{unbox } y)))) (\text{box } x) \xrightarrow{\text{CSP}} (\text{box} (\lambda x'. (\text{unbox } (\text{box } x)))) \\
(\lambda y. (\text{box} (\lambda x. (\text{unbox } y)))) (\text{box } x) \xrightarrow{\text{Lisp}} (\text{box} (\lambda x. (\text{unbox } (\text{box } x))))
\]

\[\xrightarrow{\text{CSP}} (\text{box} (\lambda x'. x)) \]

\[\xrightarrow{\text{Lisp}} (\text{box} (\lambda x.x))\]

### 1.2 Target Language

This difference in variable scoping fundamentally influences the way of unstaging translation. We unstage CSP multi-staged programs into a context calculus, while Choi et al. [4] unstaged Lisp-like multi-staged programs into a record calculus.

Record calculus is well suited as the target language for unstaging Lisp-like multi-staged programs. Since variables are bound to a specific stage, Choi et al. maintained environments for each stage and pass the appropriate one to the \( \text{unbox} \) expression. For example,

\[
\text{box} (\lambda y. (\text{unbox} ((\lambda x. \text{box} x) \text{box} y))) \rightarrow \lambda \rho. \lambda x. ((\lambda \rho. \rho \cdot x) \{ x = x \}) \rightarrow \lambda \rho'. \lambda x. x
\]

In the original program (in the leftmost), \( x \) is bound to \( \lambda x \) through \( \text{unbox} \) and \( \text{box} \). After unstaged (in the middle), \( x \) is translated to \( x \) which is connected to \( \lambda x \) through the record \( \{ x = x \} \); \( x \) is eventually bound to \( \lambda x \) (in the rightmost).

**Why Not Record Calculus** Choi et al.’s unstaging translation [4] of Lisp-like multi-staged programs does not apply directly to CSP multi-staged programs. For example,

\[
(\lambda x. \text{box} x) 0 \rightarrow (\lambda x. (\lambda \rho. \rho \cdot x)) 0 \\
\text{box} 0 \rightarrow (\lambda \rho. 0) \neq (\lambda \rho. \rho \cdot x) \rightarrow \text{box } x
\]

In general, cross-stage persistence is a fundamental obstacle for record calculus as the target language. For details, consider the example:

\[
\text{box} (\lambda y. (\text{unbox} ((\lambda x. \text{box} x) \text{box} y))) \rightarrow \text{box} (\lambda y. (\text{unbox} (\text{box} (\text{box} y)))) \\
\rightarrow \text{box} (\lambda y. (\text{box} y))
\]

**Why Context Calculus** The context calculus is the target language for our unstaging translation of the CSP multi-staged programs. Variable-capturing substitution is the key feature of context calculus used in the translation.

\[
\text{box} (\lambda x. \text{unbox} (\text{box} x)) \xrightarrow{\text{variable-capturing substitution}} (\lambda h. \text{box} (\lambda x. \text{unbox } h)) (\text{box } x)
\]

\[
\text{box} (\lambda x'. \text{unbox} (\text{box} x)) \xrightarrow{\text{capture-avoiding substitution}} \neq (\lambda h. \text{box} (\lambda x. \text{unbox } h)) (\text{box } x)
\]

In the above example, \( \text{box } x \) should be dragged outside of the enclosing \( \text{box} \) (see § 3 for details). Variable \( x \) is originally bound to \( \lambda x \) (in the upper-left), but it becomes unbound after dragged outside of \( \text{box} \).
(in the right). Without variable-capturing substitution, $x$ cannot be re-bound to $\lambda x$. See the difference between two left expressions.

Unstaging a CSP multi-staged program in this manner does not introduce additional complexity to the source program. Turing-Church thesis appeals that we can unstage it to pure lambda calculus, but it will introduce huge additional complexity. Thus, the unstaging translation would be unusable in analyzing programs. We will present an example of static analysis of context calculus in § 1.4.

1.3 Problem

For example, consider the following staged program $e$.

\[
\begin{align*}
  p &:= 0 \\
  s &:= \text{box } 0 \\
  \text{while } \text{cond} \text{ do} \\
  & \quad p := p + 2 \\
  & \quad s := \text{box } (p + \text{unbox } s) \\
  \text{done} \\
  \text{run } s
\end{align*}
\]

During the evaluation, $s$ was originally assigned to $\text{box } 0$. After iterations, $s$ becomes $\text{box } 2 + 0$ and then $\text{box } 4 + 2 + 0$. After the loop, $s$ have code templates as follows:

\[
\{ \text{box } S \mid (S \rightarrow 0 \mid 2N + S) \}
\]

In order to analyze $\text{run } s$ by conventional analysis techniques, analysis result for $s$ should be concretized to code values first. It is unrealizable, however, since $s$ has infinitely many code values in its concretization. Thus, it is necessary to find a way to detour this infinite concretization.

1.4 Our Solution

To analyze the staged program $e$, we first unstage $e$ into $e$.

\[
\begin{align*}
  p &:= 0 \\
  s &:= \lambda u_1.0 \\
  \text{while } \text{cond} \text{ do} \\
  & \quad p := p + 2 \\
  & \quad s := \delta H.\lambda u_2.(p + H \ () \circ (s); \\
  \text{done} \\
  s ()
\end{align*}
\]

The context calculus introduces variable-capturing version $\delta$ and $\circ$ of lambda abstraction and application, respectively. For example, $(\delta X.\lambda x.X) \circ (\lambda y.x)$ evaluates to $\lambda x.\lambda y.x$. Variable $x$ used at the right subexpression is captured by the lambda abstraction at the left subexpression, which is not a behavior of lambda abstraction and application (see § 2.2).

Now we consider the collecting semantics $[e]$. $[e]$ as concrete semantics, and concrete projection $\pi$.

- $[e]$: Collecting semantics $[e]$ of variables have values such as:

\[
\begin{align*}
  p & \quad \text{has} \quad 0, 2, 4, \cdots \\
  s & \quad \text{has} \quad \text{box } 0, \text{box } (2 + 0), \text{box } (4 + 2 + 0), \cdots \\
  \text{run } s & \quad \text{has} \quad 0, 2, 6, \cdots
\end{align*}
\]
- $\llbracket e \rrbracket$: Collecting semantics $\llbracket e \rrbracket$ of variables have closure values such as:

$$
\begin{align*}
    p & \text{ has } (0, \emptyset), (2, \emptyset), (4, \emptyset), \ldots \\
    s & \text{ has } (\lambda u_1.0, \emptyset), \\
        & \quad (\lambda u_2.(p_1 + H ()), \{ p_1 \mapsto 2, H \mapsto \lambda u_1.0 \}), \\
        & \quad (\lambda u_2.(p_2 + (\lambda u_2.(p_1 + H ())) ()), \{ p_2 \mapsto 4, p_1 \mapsto 2, H \mapsto \lambda u_1.0 \}), \\
    u_1 & \text{ has } ((), \emptyset) \\
    u_2 & \text{ has } ((), \emptyset) \\
    H & \text{ has } (\lambda u_1.0, \emptyset), \\
        & \quad (\lambda u_2.(p_1 + H ()), \{ p_1 \mapsto 2, H \mapsto \lambda u_1.0 \}), \\
        & \quad (\lambda u_2.(p_2 + (\lambda u_2.(p_1 + H ())) ()), \{ p_2 \mapsto 4, p_1 \mapsto 2, H \mapsto \lambda u_1.0 \}), \\
    s & \text{ has } (0, \emptyset), (2, \emptyset), (6, \emptyset), \ldots
\end{align*}
$$

- $\pi$: Projection $\pi$ is safe if it forgets extra bindings introduced in the target language and projects closure value to code expression whose unbox expression’s code are those projected from the environment.

Now we consider the abstract semantics $\llbracket e \rrbracket$, $\llbracket e \rrbracket$, and $\dot{\pi}$.

- $\llbracket e \rrbracket$: Suppose we abstract relationships among code values in terms of a regular term grammar. For example, below grammar means that only one unbox hole in the code value indexed as $u_2$ is plugged by the code value indexed as $u_1$:

$$S \to u_2(u_1)$$

- $\llbracket e \rrbracket$: As an example analysis we use the style of set-based analysis because it is easy to convey what’s going on in analysis. Any static analysis can be employed in our framework. For the translated program, we get following result:

$$
\begin{align*}
    X_p & \supseteq \{ 0, 2, 4, \ldots \} (= 2N) \\
    X_s & \supseteq \{ \lambda u_1.0, \lambda u_2.(p + H ()) \} \\
    X_{u_1} & \supseteq \{ () \} \\
    X_{u_2} & \supseteq \{ () \} \\
    X_H & \supseteq \{ \lambda u_1.0, \lambda u_2.(p + H ()) \} \\
    X_{(H ())} & \supseteq \text{A set generated by } (V_1 \to 0 | p + V_1) \\
    X_{(s ())} & \supseteq \text{A set generated by } (V_2 \to 0 | p + V_1)
\end{align*}
$$

- $\dot{\pi}$: Lastly, abstract projection $\dot{\pi}$ generates a regular term grammar from the analysis result of a context program. And $\dot{\pi}$ also forgets extra bindings as $\pi$ do. For the example, we get following results:

$$
\begin{align*}
    \dot{\pi} & \rightarrow \\
        \begin{cases}
            X_p & \supseteq \{ 0, 2, 4, \ldots \} \\
            X_s & \supseteq \{ \lambda u_1.0, \lambda u_2.(p + H ()) \} \\
            X_{(s ())} & \supseteq \text{A set generated by } (V_2 \to 0 | p + V_1)
        \end{cases}
\end{align*}
$$

Note that $S_3 \to u_1 | u_2(S_3)$ can be inferred from the set-based analysis result of $X_H$.

1.5 Comparisons

While Choi et al.’s static analysis framework [4] is for Lisp-like multi-staged programs, we focus on the different multi-staging semantics: CPS multi-staged programs. As shown in § 1.1, the two multi-staging
Fig. 1. Operational Semantics of $\lambda_S$

semantics are totally different. Choi et al. unstaged Lisp-like multi-staged programs to a record calculus, while we unstage CSP multi-staged programs to a context calculus. Two translations differ in the way of passing environments of each stage. In unstaging translation of Lisp-like multi-staged calculus, an environment at each stage does not depend on environments of any other stages. On the other hand, in unstaging translation of CSP multi-staged calculus, environments of each stage sometimes should be joined to satisfy the cross-stage persistence. Our translation and theirs both simulate source programs and introduce admin reductions.

Inoue and Taha’s erasure theorem [13] works only for call-by-name program. For call-by-value multi-staged programs, the eraser transformation fails to preserve the semantics. In addition, it would be hard to extend the erasure for call-by-value with imperative features. The eraser changes the evaluation order in call-by-value case.

1.6 Organization

Section 2 defines both the CSP multi-staged calculus $\lambda_S$ and the context calculus $\lambda_C$. Section 3 defines the unstaging translation and proves its soundness. Section 4 presents the sound static analysis, based on the projection. Section 5 presents related works. Section 6 concludes.

2 Languages

2.1 Cross-Stage Persistent Multi-Staged Calculus $\lambda_S$

CSP multi-staged calculus $\lambda_S$ is a call-by-value $\lambda$-calculus with staging constructs whose variables are cross-stage persistent.

Syntax Let $x$ range over the set of variables and $i$ range over the set of constants. The set $Expr_S$ of expressions $e$ of CSP multi-staged language is defined as follows:

$$ e ::= i \mid x \mid \lambda x.e \mid e \ e \mid \text{box} \ e \mid \text{unbox} \ e \mid \text{run} \ e $$

Expressions consist of constants, variables, abstractions, applications, boxes, unboxes, and runs. A box promotes the stage and generates a code template, while an unbox demotes the stage and escapes from a code template to be replace with another code template. A run expression executes a code template. Unlike Lisp-like multi-staged calculus, CSP multi-staged calculus admits the lift operator by $\text{lift} \ e \ =_{\text{def}} (\lambda x.\text{box} \ x) \ e$. 
Figure 1 provides a small-step call-by-value operational semantics. Judgment $e \xrightarrow{n} e'$ means $e$ is reduced to $e'$ at stage $n$. Values are defined for every stage as follows:

\[
\begin{align*}
\text{Value}_0^v &= i \mid x \mid \lambda x.e_0 \mid \text{box } v^1 \\
\text{Value}^v_n(n \geq 1) &= v^n ::= i \mid x \mid \lambda x.v^n \mid v^n v^n \\
&\quad \mid \text{box } v^{n+1} \mid \text{unbox } v^{n-1}(n \geq 2) \mid \text{run } v^n 
\end{align*}
\]

Each $n$-stage value is not reduced at stage $n$. A value at stage $n$ is also a value at stage $n+1$:

\[
\text{Value}^v_n \subseteq \text{Value}^v_{n+1}
\]

Atomic reductions are carried out only in $(\text{APP}_S)_2$, $(\text{UNB}_S)_2$ and $(\text{RUN}_S)_2$. Both $(\text{APP}_S)_3$ and $(\text{RUN}_S)_2$ are applied at stage 0, but $(\text{UNB}_S)_2$ is applied at stage 1. Rule $(\text{RUN}_S)_2$ executes the code template. Rule $(\text{UNB}_S)_2$ is similar to $(\text{RUN}_S)_2$, but unbox expression is replaced with the code template $v^1$ and immediately frozen. See our technical report [3] for the formal definition of the substitution rules.

### 2.2 Context Calculus $\lambda_C$

The context calculus $\lambda_C$ is a call-by-value $\lambda$-calculus with first-class contexts. The target language $\lambda_C$ is based on Hashimoto and Ohori [10], and restricts its semantics to call-by-value reductions. The context calculus has variable substitution for usual lambda abstractions and hole-filling substitution for hole-filling abstractions as follows:

\[
(\lambda h.\lambda x.H) x \rightarrow \lambda y.x \quad (\delta H.\lambda x.H) \circ x \rightarrow \lambda x.x
\]

**Syntax** Let $x$ range over the set $\text{Var}$ of variables, $X$ range over the set of hole variables, $\nu$ range over the set of renamers, and $i$ range over the set of constants. The set $\text{Expr}_C$ of expressions $e$ of the context calculus is defined as follows:

\[
e ::= i \mid x \mid \lambda x.e \mid e e \mid X^\nu \mid \delta X.e \mid e \circ_\nu e
\]

Hole abstraction $\delta X.e$ binds the hole $X$ in $e$, and $e \circ_\nu e$ is a hole-filling application. Renamer $\nu = \{y_1/x_1, \ldots, y_n/x_n\}$ is a partial function from $\text{Var}$ to $\text{Var}$. The renamer can easily be extended to a total function by letting $\nu(x) = x$ for all $x \notin \text{dom}(\nu)$. See [10] for more details.

**Operational Semantics** Figure 2 provides a small-step call-by-value operational semantics. Judgment $e \xrightarrow{c} e'$ means $e$ is reduced to $e'$. An expression is a value if and only if it is neither an application nor a hole filling application:

\[
\text{Value}_C \quad v ::= i \mid x \mid X \mid \lambda x.e \mid \delta X.e
\]

By $\{v/x\} e$ we mean the variable substitution of $x$ for $v$ in $e$ and by $e[v/X]$ we mean the hole substitution of $X$ for $v$ in $e$. See [3, 10] for the formal definition of $\nu$ and substitution. In a hole filling application $e_1 \circ_\nu e_2$, $e_2$'s free variables that belong to $\text{dom}(\nu)$ become bound in $e_1 \circ_\nu e_2$. For example,

\[
\{y/x\}(e \circ_\{w/x\} x) = ((y/x)e) \circ_\{w/x\} x \\
\neq ((y/x)e) \circ_\{w/x\} y
\]
the variable \( x \) is not substituted for \( y \), since \( x \) is bound to the renamer \( \{w/x\} \). During evaluation, a hole filling application \((\delta X.e_1) \odot \nu e_2\), the free variables in \( e_2 \) is at first renamed by the renamer \( \nu \) and next renamed again by the renamer \( \nu' \) associated with \( X \) in \( e_1 \). For example,

\[
(\delta X.\lambda x.X^{\{w/x\}}) \odot \{y/z\} z \rightarrow \lambda x.x \quad (\delta X.\lambda z.X^{(1)}) \odot \{y/z\} z \rightarrow \lambda x.z
\]

In the left expression, bound variable \( z \) is renamed to \( x \) during the hole filling application, and is re-bound to \( \lambda x \). In the right expression, on the other hand, \( z \) remains free.

3 Translation

We present the unstaging translation from CSP multi-staged calculus \( \lambda_S \) to the context calculus \( \lambda_C \). Our unstaging translation is related to the translation of closed codes by Davies and Pfenning [7] and the translation of Lisp-like multi-staged programs by Choi et al. [4].

Basically, we unstage a \textit{box} into lambda abstraction as follows:

\[
\text{box} \ 1 \mapsto \lambda u.1
\]

and a \textit{run} (and also a \textit{unbox}) into a lambda application with a dummy expression \((\) as follows:

\[
\text{run} (\text{box} \ 1) \mapsto (\lambda u.1) ()
\]

\[
\text{unbox} (\text{box} \ 1) \mapsto (\lambda u.1) ()
\]

Before unstaging \textit{unbox}, however, we translate \textit{unbox} in the continuation passing style. In well-formed expressions, an \textit{unbox} always occurs in a \textit{box}, and the \textit{unbox} evaluates its subexpression if it demotes to the stage 0. Recall that we translate a \textit{box} expression into a lambda abstraction, and any subexpression inside the lambda abstraction is not reduced before the enclosing lambda abstraction in the call-by-value semantics. So we have to pull out the subexpression outside of the enclosing \textit{box} by putting it into the continuation as follows:

\[
\text{box} (\cdots \text{unbox} \ e \cdots) \\ \mapsto (\lambda h.\text{box} (\cdots \text{unbox} \ h \cdots)) \ e
\]

At the same time, we have to re-bind bound variables in the subexpression pulled out. If an \textit{unboxed} expression has bound variables, they become unbound in the subexpression pulled out from the enclosing \textit{box}. In the example below,

\[
\text{box} (\lambda x.\text{unbox} \ (\text{box} \ x)) \mapsto (\lambda h.\text{box} (\lambda x.\text{unbox} \ h)) (\text{box} \ x)
\]

while the variable \( x \) was bound, it becomes unbound after translation. The translated image of the above example behaves differently than expected as follows:

\[
(\lambda h.\text{box} (\lambda x.\text{unbox} \ h)) (\text{box} \ x) \rightarrow (\lambda h.\text{box} (\lambda x.\text{unbox} \ h)) (\text{box} \ x) \\ \rightarrow (\lambda h.\text{box} (\lambda x.\text{unbox} \ h)) (\text{box} \ x)
\]

Therefore, we translate \textit{unbox} expressions to a hole abstraction and a hole-filling application to preserve bound variables. Hole-filling application ‘passes’ bound variables so that bound variables in \textit{unbox} expressions are preserved as follows:

\[
\text{box} (\cdots \text{unbox} \ e \cdots) \\ \mapsto (\delta H.\text{box} (\cdots \text{unbox} \ H \cdots)) \odot e
\]

For example, we can safely translate the above example as follows:

\[
\text{box} (\lambda x.\text{unbox} \ (\text{box} \ x)) \mapsto (\delta H.\text{box} (\lambda x.\text{unbox} \ H^{\{w/x\}})) \odot (w/x) (\text{box} \ x) \\ \rightarrow (\lambda x.\text{unbox} \ (\text{box} \ x))
\]
In the above example, \{w/x\} associated with \(\circ\) is a renamer which means that \(x\) is bound variable in (box \(x\)). Renamer \{x/w\} associated with \(H\) means that \(x\) might be bound to the \(\lambda x\). In these renamers, \(w\) is an interface variable and it bridges between the lambda binding \(\lambda x\) and the bound variable \(x\). Hole-filling applications pass bound variables through renamers and interface variables. See [10] for more details.

We should carefully determine the renamers associated with hole abstractions and hole-filling applications. Consider the following two similar \(\lambda_S\) expressions and their translation images:

\[
\begin{align*}
\text{box } (\lambda x.\text{unbox (box } x\text{)}) &\rightarrow (\delta H.\text{box} (\lambda x.\text{unbox } H^{(x/w)}) \circ_{\{w/x\}} (\text{box } x)) \\
\lambda x.\text{box} (\text{unbox (box } x\text{)}) &\rightarrow \lambda x.(\delta H.\text{box} (\text{unbox } H^{(1)})) \circ_{\{\text{\ldots}\}} (\text{box } x)
\end{align*}
\]

In the first example, the variable \(x\) is not directed bound to \(\lambda x\), so we have to bind it with renamers. However, in the second example, the variable \(x\) is already directly bound to \(\lambda x\), so we must not bind it with renamers. In order to determine the correct renamers, we have to find which bound variables in unboxed expressions become free variables after translation.

### 3.1 Translation Rules

Figure 3 presents the inference rule of unstaging translation. Judgment \(R \vdash e \rightarrow (\xi, K)\) means that under an environment stack \(R\), an expression \(e\) of \(\lambda_S\) is translated into an expression \(\xi\) of \(\lambda_C\) with a continuation stack \(K\). Our translation rules are similar to [4] except the rule (TVAR) and (TUNB) and the definitions of environment and continuation (counterpart of context of [4]).

As opposed to Choi et al. [4], a variable remains as is to support cross-stage persistence. In \(\lambda_S\), variables of any stage is allowed to be used in any other stages, that is, behavior of variables are not restricted by staging constructs. Thus, the translated image of the variable is the same with the original variable. For example,

\[
\lambda x.\text{box } x\xrightarrow{0} (\lambda x.\lambda u.x) 0 \\
\text{box } 0\xrightarrow{} \lambda u.0
\]

Another big difference between [4] and our translation is that in (TUNB) the last (rightmost) environment \(r_n\) is accumulated with the right before environment \(r_{n-1}\), while [4] discards the last environment \(r_n\). The newly generated environment \((r_{n-1}; r_n)^1\) is, once again, accumulated with the right before environment \(r_{n-2}\) when we meet another unbox as the translation goes further inside \(e\). The accumulation is intended to correctly determine the renamers for each unbox expression. If a variable is once defined, it should be visible for all lower stages in the translation. For example, consider the translation of the following program (note that we write \([x_1, x_2]\) for an arbitrary expression including the variables \(x_1\) and \(x_2\), for the simplicity):

\[
\text{box } (\lambda x_1, (\text{box} (\lambda x_2, (\text{unbox } [x_1, x_2])))
\]

The above program is unstaged as follows:

\[
(\delta H_1.\lambda u_1.\lambda x_1.((\delta H_2.\lambda u_2.\lambda x_2. (H_2^{x_2})(\lambda x_1)) \circ_{x_2} (H_2^{x_1}(1))) \circ_{u_1} [x_1, x_2])
\]

where \(u_1 = \{w_1/x_1, w_0/x_2\}\) and \(u_2 = \{w_0/x_2\}\). Note that \(u_1\) is not \(\{w_0/x_1\}\) but \(\{w_1/x_1, w_0/x_2\}\). Recall that the variables \(x_1\) and \(x_2\) that are occurred freely in \([x_1, x_2]\) should be bound again to \(\lambda x_1\) and \(\lambda x_2\) after the hole filling applications. While \(x_2\) is eventually put back into \(H_1\) and again into \(H_2\), \(u_1\) should have the entry for \(x_2\) since \(x_2\) is bound at \(H_1\). Note that the renamer \(u_2(=\{w_0/x_2\})\) binds \(x_2\) at \(H_1\) as well as the lambda abstraction \(\lambda x_2\) binds \(x_2\) at \(H_2\).

\(^1\) We write \((r_1; r_2)\) for concatenation of two lists \(r_1\) and \(r_2\)
Definitions

Environment $r ::= \bot \mid r; x$

Environment Stack $R ::= r \mid R, r$

Continuation $\kappa ::= (\delta H, [\cdot]) \circ_\nu e \mid (\delta H, \kappa) \circ_\nu e$

Continuation Stack $K ::= \bot \mid K, \kappa$

Translation

\begin{align*}
(TCON) & \quad R \vdash i \mapsto (i, \bot) \\
(TVAR) & \quad R \vdash x \mapsto (x, \bot)
\end{align*}

\begin{align*}
(TABS) & \quad R, (r_n; x) \vdash e \mapsto (\langle \epsilon, K \rangle) \\
 & \quad R, r_n \vdash \lambda x. e \mapsto (\langle \lambda x. \epsilon, K \rangle)
\end{align*}

\begin{align*}
(TAPP) & \quad R \vdash e_1 \mapsto (e_1, K_1) \\
 & \quad R \vdash e_2 \mapsto (e_2, K_2)
\end{align*}

\begin{align*}
(TBOX) & \quad R, \bot \vdash \langle \epsilon, (K, \kappa) \rangle \quad \text{new } u \\
 & \quad R \vdash \text{box } e \mapsto (\langle \epsilon, K \rangle)
\end{align*}

\begin{align*}
(TUNB) & \quad R, (r_{n-1}; r_n) \vdash e \mapsto (\langle \epsilon, K \rangle) \\
 & \quad R, r_{n-1}, r_n \vdash \text{unbox } e \mapsto (H^{n-1}_\nu \cdot (K, ((\delta H, [\cdot]) \circ_\nu \epsilon))) \\
 & \quad R, r_{n-1}, r_n \vdash e \mapsto (\langle \epsilon, K \rangle) \quad \text{new } H
\end{align*}

\begin{align*}
(TRUN) & \quad R \vdash e \mapsto \langle \epsilon, K \rangle \quad \text{new } h \\
 & \quad R \vdash \text{run } e \mapsto \langle \text{let } h = \epsilon \text{ in } \epsilon, K \rangle
\end{align*}

Renamer Inverse

\[
\{y_1/x_1, \ldots, y_n/x_n\}^{-1} = \{x_1/y_1, \ldots, x_n/y_n\}
\]

Continuation Stack Merge

\[
\bot \bowtie K = K \\
K \bowtie \bot = K \\
(K_1, \kappa_1) \bowtie (K_2, \kappa_2) = (K_1 \bowtie K_2, (\kappa_1[\kappa_2])
\]

Fig. 3. Translation from $\lambda_S$ to $\lambda_C$
Definitions

Hole Environment $H \in \text{HoleVar}$

Translation

<table>
<thead>
<tr>
<th>ICON</th>
<th>$H \vdash i \mapsto i$</th>
<th>(ivar)</th>
<th>$H \vdash x \mapsto x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>IABS</td>
<td>$H \vdash e \mapsto e$</td>
<td>(iapp)</td>
<td>$H \vdash e_1, e_2 \neq \emptyset \mapsto e_1, e_2$</td>
</tr>
<tr>
<td>ICTX</td>
<td>$H \cup {H : e'_1} \vdash e \mapsto e$</td>
<td>(ibox)</td>
<td>$H \vdash \lambda u.e \mapsto \text{box } e$</td>
</tr>
<tr>
<td>IUNB</td>
<td>$H \vdash \delta H.e \mapsto e$</td>
<td>(irun)</td>
<td>$H \vdash \lambda u.e \mapsto \text{run } e$</td>
</tr>
</tbody>
</table>

Continuation Cumulation Operator

$\top = \emptyset$

$\bot = \emptyset$

$\langle K, \kappa \rangle = K \cup \kappa$

$\delta H.e \circ \{H: e\} = \kappa \cup \{H : e\}$

Fig. 4. Inverse Translation from $\lambda_C$ to $\lambda_S$

Also, the definitions of environment and continuation are slightly different: environment is a list of variables instead of a record, and continuation consists of a hole variable and a hole filling application instead of the usual variable and application.

Without loss of generality, we assume that for any expression $e$ of $\lambda_S$, all bound variables in $e$ are distinct. If variables are duplicated in a reduction, we systematically rename the duplicated variables to fresh variables. See [3] for more details.

3.2 Semantics Preservation

We prove our translation is semantics-preserving. Given a program, its translated image by the translation rules in Fig. 3 preserves the semantics of the original program. We prove that not only the final value but also all reduction steps are preserved.

For stating the reduction preservation property, we first introduce the administrative reduction [4, 18].

Definition 1 (Admin Reduction). Administrative reduction of an expression is a congruence closure of the following rule:

$$(\lambda u.e) O \xrightarrow{A} \{O/u\} e$$

We write $e \xrightarrow{A} e'$ for the reflexive, transitive closure of $\xrightarrow{A}$. We write $e \xrightarrow{C,A} e'$ for the reduction $\xrightarrow{C}$ followed by zero or more reductions $\xrightarrow{A}$ until no more reduction $\xrightarrow{A}$ is possible.

Using the administrative reduction, we finally state the simulation theorem of the semantics preservation. See [3] for the complete proof of Theorem 1.

Theorem 1 (Simulation). Let $e, e' \in \text{Expr}_S$ and $e \xrightarrow{0} e'$. Let $\emptyset \vdash e \mapsto (e, \bot)$ and $\emptyset \vdash e' \mapsto (e', \bot)$. Then $e \xrightarrow{C,A} e'$. Furthermore, If $e \in \text{Value}_S^0$ then $e \in \text{Value}_C^0$.

3.3 Inverse Translation

Figure 4 presents an inverse translation from $\lambda_C$ to $\lambda_S$. We almost identically adopt the inverse translation of Choi et al. [4] The inverse translation judgment $H \vdash e \mapsto e$ means that a $\lambda_C$ expression $e$
is translated back to a $\lambda_S$ expression $e$ under hole environment $H$. A hole environment is designed to restore dragged unbox subexpressions. Rule (IUNB) uses the hole environment, while rule (ICTX) stores dragged unbox subexpression in the hole environment. Note that only one hole environment sufficient for inverse translation as opposed to the forward translation since all hole variables are fresh.

By inverse translation we mean that if we translate a $\lambda_S$ expression to $\lambda_C$ expression, and then inversed it back to $\lambda_S$, then the result is the same with the input. Together with Theorem 1, Theorem 2 means that $\lambda_S$ is simulated by $\lambda_C$.

\[
e \xrightarrow{0} e' \implies e \xrightarrow{c,A^*} e'
\]

**Theorem 2 (Inverse Translation).** Let $e$ be a $\lambda_S$ expression and $R$ be an environment stack. If $R \vdash e \mapsto (e, K)$ then $K \vdash e \xmapsto{} e$.

4 Analysis

4.1 Static Analysis of the Context Calculus

We present a set-based analysis [12, 11] on the context calculus, as shown in § 1.4. In fact, variable-capturing substitution in the context calculus does not complicate during the set-based analysis, since all variables are assumed to be distinct from each other. We add a new rule for constructing set constraints of the hole-filling application, which is similar to that of the usual lambda application. See [3] for details.

4.2 Projection

We analyze the translated programs and project the analysis result back to the source language. In terms of abstract interpretation framework, this procedure results in a sound static analysis. Choi et al. [4] presented Theorem 3 for sound projection in the static analysis framework for Lisp-like multi-staged programs via unstaging translation. Since the condition is language-independent, it also applies to our framework.

**Theorem 3 (Sound Projection).** Let $e$ and $c$ be, respectively, a staged program and its translated unstaged version. If $J_e \subseteq \pi J_e$ and $\alpha \circ \pi \circ \gamma \subseteq \hat{\pi}$ then $\alpha J_e \subseteq \hat{\pi} J_e$.

5 Related Work

**Translation** For comparisons to Choi et al. [4] and Inoue and Taha [13], see § 1.5. Davies and Pfenning [7] presented a translation between implicit and explicit modal lambda calculus. The translation makes the evaluation order explicit. However, their staged calculus does not support open code templates.

Hashimoto and Ohori [10] presented a typed context calculus, our target language. To design a type system for the context calculus, they introduced renamers for hole variables and hole-filling applications.

**Analysis** In terms of Cousot and Cousot’s abstract interpretation framework [5], we presented a static analysis framework by which a sound static analysis of CSP multi-staged language is derived from that of context calculus. Static analysis of multi-staged calculus was not widely studied. Kamin et al. [15] presented an interleaving analysis of multi-staged calculus which is both static and dynamic. Our analysis via translation is completely static.

Taha presented a type system for CSP multi-staged language [24], but it has a problem on open code templates. To solve this, Taha introduced a type system based on environment classifiers [22] to loose a restriction on closed codes. On the other hand, Kim et al. presented a polymorphic type system for Lisp-like multi-staged languages [16].
6 Conclusion

We present the unstaging translation and prove that it preserves the small-step operational semantics by step-by-step simulation. This unstaging translation enables static analysis by 1) unstaging the source program, 2) analyzing the unstaged program using conventional static analysis techniques, and 3) projecting the analysis result back to the source language. Our translation supports all fundamental CSP multi-staged features: code substitution, code execution, and cross-stage persistence.

References