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-qutation 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,

$$(\lambda x.\texttt{box}\ x)\ 0 \xrightarrow{\texttt{CSP}} \texttt{box}\ 0 \qquad (\lambda x.\texttt{box}\ x)\ 0 \xrightarrow{\texttt{Lisp}} \texttt{box}\ x$$

In CSP multi-staged language (on the left) x in `box` is bound to λx, but in Lisp-like multi-staged language (on the right) x in `box` is not bound to λx. CSP multi-staged language supports only capture-avoiding substitution. On the other hand, Lisp supports both variable-capturing and capture-avoiding substitution. For example,

$$\begin{aligned}(\lambda y.(\texttt{box}\ (\lambda x.(\texttt{unbox}\ y))))\ (\texttt{box}\ x) &\xrightarrow{\texttt{CSP}} (\texttt{box}\ (\lambda x'.(\texttt{unbox}\ (\texttt{box}\ x)))) \\ &\xrightarrow{\texttt{CSP}} (\texttt{box}\ (\lambda x'.x))\end{aligned}$$

$$\begin{aligned}(\lambda y.(\texttt{box}\ (\lambda x.(\texttt{unbox}\ y))))\ (\texttt{box}\ x) &\xrightarrow{\texttt{Lisp}} (\texttt{box}\ (\lambda x.(\texttt{unbox}\ (\texttt{box}\ x)))) \\ &\xrightarrow{\texttt{Lisp}} (\texttt{box}\ (\lambda x.x))\end{aligned}$$

1.2 Target Language

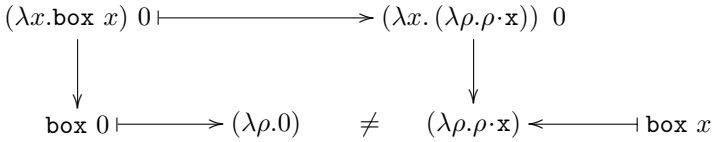
This difference in variable scoping fundamentally influences the way of unstaging translation. We unstage CSP multi-staged programs into a context calculus, while Choi et al. [4] unstaged Lisp-like multi-staged programs into a record calculus.

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 `unbox` expression. For example,

$$\texttt{box}\ (\lambda x.(\texttt{unbox}\ \underline{(\texttt{box}\ x)})) \mapsto \lambda\rho'.\lambda x.(\underline{(\lambda\rho.\rho{\cdot}\texttt{x})}\ \{\texttt{x} = x\}) \longrightarrow \lambda\rho'.\lambda x.x$$

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

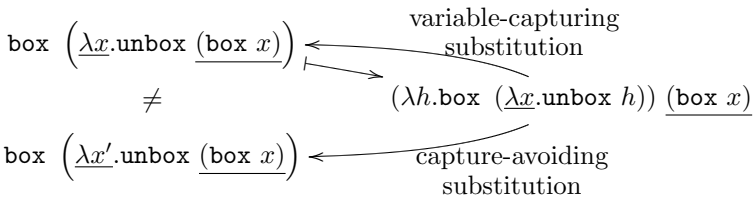
Why Not Record Calculus Choi et al.'s unstaging translation [4] of Lisp-like multi-staged programs does not apply directly to CSP multi-staged programs. For example,



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

$$\begin{aligned}\texttt{box}\ (\lambda y.(\texttt{unbox}\ ((\lambda x.\texttt{box}\ x)\ \ \texttt{box}\ y))) &\longrightarrow \texttt{box}\ (\lambda y.(\texttt{unbox}\ (\texttt{box}\ (\texttt{box}\ y)))) \\ &\longrightarrow \texttt{box}\ (\lambda y.(\texttt{box}\ y)))\end{aligned}$$

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



In the above example, `box` x should be dragged outside of the enclosing `box` (see § 3 for details). Variable x is originally bound to λx (in the upper-left), but it becomes unbound after dragged outside of `box`

(in the right). Without variable-capturing substitution, x cannot be re-bound to λx. 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.

$$
\begin{array}{ll}
p := 0 & \\
s := \texttt{box}\ 0 & \text{(* indexed as } u_1 \text{ *)} \\
\texttt{while}\ \mathit{cond}\ \texttt{do} & \\
\quad p := p + 2 & \\
\quad s := \texttt{box}\ (p + \texttt{unbox}\ s) & \text{(* indexed as } u_2 \text{ *)} \\
\texttt{done} & \\
\texttt{run}\ s &
\end{array}
$$

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

$$\{\texttt{box}\ S \mid (S \to 0 \mid 2\mathbb{N} + S)\}$$

In order to analyze `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 $\underline{e}$.

$$
\begin{array}{l}
p := 0 \\
s := \lambda u_1.0 \\
\texttt{while}\ \mathit{cond}\ \texttt{do} \\
\quad p := p + 2 \\
\quad s := \delta H.\lambda u_2.(p + H\ ()) \odot (s); \\
\texttt{done} \\
s\ \texttt{()}
\end{array}
$$

The context calculus introduces variable-capturing version δ and $\odot$ of lambda abstraction and application, respectively. For example, $(\delta X.\lambda x.X) \odot (\lambda y.x)$ 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 $[\![e]\!]$, $[\![\underline{e}]\!]$ as concrete semantics, and concrete projection π.

– $[\![e]\!]$: Collecting semantics $[\![e]\!]$ of variables have values such as:

$$
\begin{array}{lll}
p & \text{has} & 0, 2, 4, \cdots \\
s & \text{has} & \texttt{box}\ 0, \texttt{box}\ (2 + 0), \texttt{box}\ (4 + 2 + 0), \cdots \\
\texttt{run}\ s & \text{has} & 0, 2, 6, \cdots
\end{array}
$$

– $[\![\underline{e}]\!]$: Collecting semantics $[\![\underline{e}]\!]$ of variables have closure values such as:

$$\begin{array}{ll} p & \text{has } \langle 0, \varnothing\rangle, \langle 2, \varnothing\rangle, \langle 4, \varnothing\rangle, \cdots \\ s & \text{has } \langle \lambda u_1.0, \varnothing\rangle, \\ & \quad \langle \lambda u_2.(p_1 + H\ ()), \{p_1 \mapsto 2, H \mapsto \lambda u_1.0\}\rangle, \\ & \quad \langle \lambda u_2.(p_2 + (\lambda u_2.(p_1 + H\ ()))\ ()), \{p_2 \mapsto 4, p_1 \mapsto 2, H \mapsto \lambda u_1.0\}\rangle, \\ & \quad \cdots \\ u_1 & \text{has } \langle (), \varnothing\rangle \\ u_2 & \text{has } \langle (), \varnothing\rangle \\ H & \text{has } \langle \lambda u_1.0, \varnothing\rangle, \\ & \quad \langle \lambda u_2.(p_1 + H\ ()), \{p_1 \mapsto 2, H \mapsto \lambda u_1.0\}\rangle, \\ & \quad \langle \lambda u_2.(p_2 + (\lambda u_2.(p_1 + H\ ()))\ ()), \{p_2 \mapsto 4, p_1 \mapsto 2, H \mapsto \lambda u_1.0\}\rangle, \\ & \quad \cdots \\ s\ () & \text{has } \langle 0, \varnothing\rangle, \langle 2, \varnothing\rangle, \langle 6, \varnothing\rangle, \cdots \end{array}$$

– π: Projection π 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 $[\![\hat{e}]\!]$, $[\![\underline{\hat{e}}]\!]$, and $\hat{\pi}$.

– $\hat{[\![\underline{e}]\!]}$: 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:

$$S \to u_2(u_1)$$

– $\hat{[\![\underline{e}]\!]}$: 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:

$$\begin{array}{rl} \mathcal{X}_p & \supseteq \{0, 2, 4, \cdots\} (= 2\mathbb{N}) \\ \mathcal{X}_s & \supseteq \{\lambda u_1.0, \lambda u_2.(p + H\ ())\} \\ \mathcal{X}_{u_1} & \supseteq \{()\} \\ \mathcal{X}_{u_2} & \supseteq \{()\} \\ \mathcal{X}_H & \supseteq \{\lambda u_1.0, \lambda u_2.(p + H\ ())\} \\ \mathcal{X}_{(H\ ())} & \supseteq A \textit{ set generated by } (V_1 \to 0 \mid p + V_1) \\ \mathcal{X}_{(s\ ())} & \supseteq A \textit{ set generated by } (V_2 \to 0 \mid p + V_1) \end{array}$$

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

$$\begin{cases} \mathcal{X}_p \supseteq \{0, 2, 4, \cdots\} \\ \mathcal{X}_s \supseteq \{\lambda u_1.0, \lambda u_2.(p + H\ ())\} \\ \mathcal{X}_{(s\ ())} \supseteq A \textit{ set generated by } (V_2 \to 0 \mid p + V_1) \end{cases}$$

$$\overset{\hat{\pi}}{\longmapsto} \begin{cases} p \text{ has } 0, 2, 4, \cdots \\ s \text{ has } \begin{cases} S_2 \to u_2(S_3) \\ S_3 \to u_1 \mid u_2(S_3) \end{cases} \\ \texttt{run}\ s \text{ has } \textit{values generated by } (V \to 0 \mid p + V) \end{cases}$$

Note that $S_3 \to u_1 \mid u_2(S_3)$ can be inferred from the set-based analysis result of $\mathcal{X}_H$.

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

$$(\mathtt{APP}_{\mathcal{S}})\ \frac{e_1 \xrightarrow{n} e_1'}{e_1\ e_2 \xrightarrow{n} e_1'\ e_2} \qquad \frac{e \xrightarrow{n} e'}{v^n\ e \xrightarrow{n} v^n\ e'} \qquad \frac{}{(\lambda x.e^0)\ v^0 \xrightarrow{0} [x \mapsto v^0]e^0}$$

$$(\mathtt{ABS}_{\mathcal{S}})\ \frac{e \xrightarrow{n+1} e'}{\lambda x.e \xrightarrow{n+1} \lambda x.e'} \qquad (\mathtt{BOX}_{\mathcal{S}})\ \frac{e \xrightarrow{n+1} e'}{\mathtt{box}\ e \xrightarrow{n} \mathtt{box}\ e'}$$

$$(\mathtt{UNB}_{\mathcal{S}})\ \frac{e \xrightarrow{n} e'}{\mathtt{unbox}\ e \xrightarrow{n+1} \mathtt{unbox}\ e'} \qquad \frac{}{\mathtt{unbox}\ (\mathtt{box}\ v^1) \xrightarrow{1} v^1}$$

$$(\mathtt{RUN}_{\mathcal{S}})\ \frac{e \xrightarrow{n} e'}{\mathtt{run}\ e \xrightarrow{n} \mathtt{run}\ e'} \qquad \frac{}{\mathtt{run}\ (\mathtt{box}\ v^1) \xrightarrow{0} v^1}$$

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

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 environmemts 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 progras, 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 $\lambda_{\mathcal{S}}$ and the context calculus $\lambda_{\mathcal{C}}$. 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 $\lambda_{\mathcal{S}}$

CSP multi-staged calculus $\lambda_{\mathcal{S}}$ is a call-by-value λ-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 $\mathit{Expr}_{\mathcal{S}}$ of expressions e of CSP multi-staged language is defined as follows:

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

Expressions consist of constants, variables, abstractions, applications, `box`es, `unbox`es, and `run`s. 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 $\mathtt{lift}\ e =_{\mathtt{def}} (\lambda x.\mathtt{box}\ x)\ e$.

$$\frac{e_1 \xrightarrow{\mathcal{C}} e_1'}{e_1\ e_2 \xrightarrow{\mathcal{C}} e_1'\ e_2} \qquad \frac{e \xrightarrow{\mathcal{C}} e'}{v\ e \xrightarrow{\mathcal{C}} v\ e'} \qquad \frac{}{(\lambda x.e)\ v \xrightarrow{\mathcal{C}} \{v/x\}e}$$

$$\frac{e_1 \xrightarrow{\mathcal{C}} e_1'}{e_1 \odot_\nu e_2 \xrightarrow{\mathcal{C}} e_1' \odot_\nu e_2} \qquad \frac{e \xrightarrow{\mathcal{C}} e'}{v \odot_\nu e \xrightarrow{\mathcal{C}} v \odot_\nu e'} \qquad \frac{}{(\delta X.e) \odot_\nu v \xrightarrow{\mathcal{C}} (e[X^\nu/X])[v/X]}$$

Fig. 2. Operational Semantics of $\lambda_{\mathcal{C}}$

Operational Semantics 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:

$$\begin{array}{ll} \mathit{Value}^0_{\mathcal{S}} & v^0 ::= i \mid x \mid \lambda x.e^0 \mid \texttt{box}\ v^1 \\ \mathit{Value}^n_{\mathcal{S}} (n \geq 1) & v^n ::= i \mid x \mid \lambda x.v^n \mid v^n\ v^n \\ & \quad \mid \texttt{box}\ v^{n+1} \mid \texttt{unbox}\ v^{n-1} (n \geq 2) \mid \texttt{run}\ v^n \end{array}$$

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

$$\mathit{Value}^n_{\mathcal{S}} \subseteq \mathit{Value}^{n+1}_{\mathcal{S}}$$

Atomic reductions are carried out only in $(\texttt{APP}_{\mathcal{S}})_3$, $(\texttt{UNB}_{\mathcal{S}})_2$ and $(\texttt{RUN}_{\mathcal{S}})_2$. Both $(\texttt{APP}_{\mathcal{S}})_3$ and $(\texttt{RUN}_{\mathcal{S}})_2$ are applied at stage 0, but $(\texttt{UNB}_{\mathcal{S}})_2$ is applied at stage 1. Rule $(\texttt{RUN}_{\mathcal{S}})_2$ executes the code template. Rule $(\texttt{UNB}_{\mathcal{S}})_2$ is similar to $(\texttt{RUN}_{\mathcal{S}})_2$, 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 $\lambda_{\mathcal{C}}$

The context calculus $\lambda_{\mathcal{C}}$ is a call-by-value λ-calculus with first-class contexts. The target language $\lambda_{\mathcal{C}}$ 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:

$$(\lambda h.\underline{\lambda x}.h)\ x \longrightarrow \underline{\lambda y}.x \qquad\qquad (\delta H.\underline{\lambda x}.H) \odot x \longrightarrow \underline{\lambda x}.x$$

Syntax Let x range over the set Var of variables, X range over the set of hole variables, ν range over the set of renamers, and i range over the set of constants. The set $\mathit{Expr}_{\mathcal{C}}$ of expressions e of the context calculus is defined as follows:

$$e ::= i \mid x \mid \lambda x.e \mid e\ e \mid X^\nu \mid \delta X.e \mid e \odot_\nu e$$

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

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

$$\mathit{Value}_{\mathcal{C}} \quad v ::= i \mid x \mid X \mid \lambda x.e \mid \delta X.e$$

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 $\overline{\nu}$ and substitution. In a hole filling application $e_1 \odot_\nu e_2$, e_2's free variables that belong to $\mathit{dom}(\nu)$ become bound in $e_1 \odot_\nu e_2$. For example,

$$\begin{aligned} \{y/x\}(e \odot_{\{w/x\}} \underline{x}) &= (\{y/x\}e) \odot_{\{w/x\}} \underline{x} \\ &\neq (\{y/x\}e) \odot_{\{w/x\}} y \end{aligned}$$

the variable $\underline{x}$ is not substituted for y, since $\underline{x}$ 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 ν and next renamed again by the renamer ν' associated with X in e_1. For example,

$$(\delta X.\lambda x.X^{\{x/y\}}) \odot_{\{y/z\}} z \longrightarrow \lambda x.x \qquad (\delta X.\lambda x.X^{\{\}}) \odot_{\{\}} z \longrightarrow \lambda x.z$$

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

3 Translation

We present the unstaging translation from CSP multi-staged calculus λ_S to the context calculus λ_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 `box` into lambda abstraction as follows:

$$\texttt{box}\ 1 \mapsto \lambda u.1$$

and a `run` (and also a `unbox`) into a lambda application with a dummy expression `()` as follows:

$$\begin{aligned}\texttt{run}\ (\texttt{box}\ 1) &\mapsto (\lambda u.1)\ \texttt{()}\\ \texttt{unbox}\ (\texttt{box}\ 1) &\mapsto (\lambda u.1)\ \texttt{()}\end{aligned}$$

Before unstaging `unbox`, however, we translate `unbox` in the continuation passing style. In well-formed expressions, an `unbox` always occurs in a `box`, and the `unbox` evaluates its subexpression if it demotes to the stage 0. Recall that we translate a `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 `box` by putting it into the continuation as follows:

$$\begin{aligned}&\texttt{box}\ (\cdots \texttt{unbox}\ e \cdots)\\ \mapsto\ &(\lambda h.\texttt{box}\ (\cdots \texttt{unbox}\ h \cdots))\ e\end{aligned}$$

At the same time, we have to re-bind bound variables in the subexpression pulled out. If an `unbox`ed expression has bound variables, they become unbound in the subexpression pulled out from the enclosing `box`. In the example below,

$$\texttt{box}\ (\lambda x.\texttt{unbox}\ (\texttt{box}\ x)) \mapsto (\lambda h.\texttt{box}\ (\lambda x.\texttt{unbox}\ h))\ \underline{(\texttt{box}\ x)}$$

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

$$\begin{aligned}(\lambda h.\texttt{box}\ (\lambda x.\texttt{unbox}\ h))\ \underline{(\texttt{box}\ x)} &\not\longrightarrow \texttt{box}\ (\lambda x.\texttt{unbox}\ \underline{(\texttt{box}\ x)})\\ &\longrightarrow \texttt{box}\ (\lambda y.\texttt{unbox}\ \underline{(\texttt{box}\ x)})\end{aligned}$$

Therefore, we translate `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 `unbox` expressions are preserved as follows:

$$\begin{aligned}&\texttt{box}\ (\cdots \texttt{unbox}\ e \cdots)\\ \mapsto\ &(\delta H.\texttt{box}\ (\cdots \texttt{unbox}\ H \cdots)) \odot e\end{aligned}$$

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

$$\begin{aligned}\texttt{box}\ (\lambda x.\texttt{unbox}\ (\texttt{box}\ x)) &\mapsto (\delta H.\texttt{box}\ (\lambda x.\texttt{unbox}\ H^{\{x/w\}})) \odot_{\{w/x\}} \underline{(\texttt{box}\ x)}\\ &\longrightarrow \texttt{box}\ (\lambda x.\texttt{unbox}\ \underline{(\texttt{box}\ x)})\end{aligned}$$

In the above example, $\{w/x\}$ associated with $\odot$ is a *renamer* which means that x is bound variable in ($\texttt{box}\ x$). Renamer $\{x/w\}$ associated with H means that x might be bound to the λx. In these renamers, w is an *interface variable* and it bridges between the lambda binding λx 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 $\lambda_{\mathcal{S}}$ expressions and their translation images:

$$\texttt{box}\ (\underline{\lambda x.\texttt{unbox}\ (\texttt{box}\ x)}) \mapsto (\delta H.\texttt{box}\ (\underline{\lambda x.\texttt{unbox}\ H^{\{x/w\}}}))\ \odot_{\{w/x\}}\ (\texttt{box}\ x)$$
$$\underline{\lambda x.\texttt{box}\ (\texttt{unbox}\ (\texttt{box}\ x))} \mapsto \underline{\lambda x.(\delta H.\texttt{box}\ (\texttt{unbox}\ H^{\{\quad\}}))\ \odot_{\{\quad\}}\ (\texttt{box}\ x)}$$

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

3.1 Translation Rules

Figure 3 presents the inference rule of unstaging translation. Judgment $R \vdash e \mapsto (\underline{e}, K)$ means that under an environment stack R, an expression e of $\lambda_{\mathcal{S}}$ is translated into an expression $\underline{e}$ of $\lambda_{\mathcal{C}}$ 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 $\lambda_{\mathcal{S}}$, 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,

$$\begin{array}{ccc} (\lambda x.\texttt{box}\ x)\ 0 & \longmapsto & (\lambda x.\lambda u.x)\ 0 \\ \downarrow & & \downarrow \\ \texttt{box}\ 0 & \longmapsto & \lambda u.0 \end{array}$$

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 $\boxed{x_1\ x_2}$ for an arbitrary expression including the variables x_1 and x_2, for the simplicity):

$$\texttt{box}\ (\lambda x_1.(\texttt{box}\ (\lambda x_2.(\texttt{unbox}\ (\texttt{unbox}\ \boxed{x_1\ x_2}\,)))))$$

The above program is unstaged as follows:

$$\Big(\delta H_1.\lambda u_1.\lambda x_1.\underline{((\delta H_2.\lambda u_2.\lambda x_2.(H_2^{\nu_2^{-1}}\ ()))\ \odot_{\nu_2}\ (H_1^{\nu_1^{-1}}\ ()))}\Big)\ \odot_{\nu_1}\ \boxed{x_1\ x_2}$$

where $\nu_1 = \{w_1/x_1, w_0/x_2\}$ and $\nu_2 = \{w_0/x_2\}$. Note that ν_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 $\boxed{x_1\ x_2}$ should be bound again to λx_1 and λx_2 after the hole filling applications. While x_2 is eventually put back into H_1 and again into H_2, ν_1 should have the entry for x_2 since x_2 is bound at H_1. Note that the renamer $\nu_2 (= \{w_0/x_2\})$ binds x_2 at H_1 as well as the lambda abstraction λx_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

$$\begin{array}{ll} \textit{Environment} & r ::= \bot \mid r; x \\ \textit{Environment Stack} & R ::= r \mid R, r \end{array}$$

$$\begin{array}{ll} \textit{Continuation} & \kappa ::= (\delta H.[\cdot]) \odot_\nu e \mid (\delta H.\kappa) \odot_\nu e \\ \textit{Continuation Stack} & K ::= \bot \mid K, \kappa \end{array}$$

Translation

$$(\texttt{TCON}) \quad R \vdash i \mapsto (i, \bot) \qquad (\texttt{TVAR}) \quad R \vdash x \mapsto (x, \bot)$$

$$(\texttt{TABS}) \quad \frac{R, (r_n; x) \vdash e \mapsto (\underline{e}, K)}{R, r_n \vdash \lambda x.e \mapsto (\lambda x.\underline{e}, K)}$$

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

$$(\texttt{TBOX}) \quad \frac{R, \bot \vdash e \mapsto (\underline{e}, (K, \kappa)) \qquad \texttt{new}\ u}{R \vdash \texttt{box}\ e \mapsto (\kappa[\lambda u.\underline{e}], K)} \quad \frac{R, \bot \vdash e \mapsto (\underline{e}, \bot) \qquad \texttt{new}\ u}{R \vdash \texttt{box}\ e \mapsto (\lambda u.\underline{e}, \bot)}$$

$$(\texttt{TUNB}) \quad \frac{\begin{array}{c} R, (r_{n-1}; r_n) \vdash e \mapsto (\underline{e}, K) \\ r_n = x_k; \cdots; x_0 \qquad \nu = \{w_k/x_k, \cdots, w_0/x_0\} \qquad \texttt{new}\ H \end{array}}{R, r_{n-1}, r_n \vdash \texttt{unbox}\ e \mapsto (H^{\nu^{-1}}\ \texttt{()}, (K, ((\delta H.[\cdot]) \odot_\nu \underline{e})))}$$

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

Renamer Inverse

$$\{y_1/x_1, \cdots, y_n/x_n\}^{-1} = \{x_1/y_1, \cdots, x_n/y_n\}$$

Continuation Stack Merge

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

Fig. 3. Translation from λ_S to λ_C

Definitions

$$\textit{Hole Environment}\ \mathcal{H} \in \textit{HoleVar} \xrightarrow{\texttt{fin}} \textit{Expr}_{\mathcal{C}}$$

Translation

$$(\texttt{ICON})\quad \mathcal{H} \vdash i \rightarrowtail i \qquad (\texttt{IVAR})\quad \mathcal{H} \vdash x \rightarrowtail x$$

$$(\texttt{IABS})\quad \frac{\mathcal{H} \vdash \underline{e} \rightarrowtail e}{\mathcal{H} \vdash \lambda x.\underline{e} \rightarrowtail \lambda x.e} \qquad (\texttt{IAPP})\quad \frac{\mathcal{H} \vdash \underline{e_i} \rightarrowtail e_i \quad \underline{e_2} \neq \texttt{()}}{\mathcal{H} \vdash \underline{e_1}\ \underline{e_2} \rightarrowtail e_1\ e_2}$$

$$(\texttt{ICTX})\quad \frac{\mathcal{H} \cup \{H : \underline{e'}\} \vdash \underline{e} \rightarrowtail e}{\mathcal{H} \vdash (\delta H.\underline{e}) \odot_\nu \underline{e'} \rightarrowtail e} \qquad (\texttt{IBOX})\quad \frac{\mathcal{H} \vdash \underline{e} \rightarrowtail e}{\mathcal{H} \vdash \lambda u.\underline{e} \rightarrowtail \texttt{box}\ e}$$

$$(\texttt{IUNB})\quad \frac{\mathcal{H} \vdash \mathcal{H}(H) \rightarrowtail e}{\mathcal{H} \vdash H\ \texttt{()} \rightarrowtail \texttt{unbox}\ e} \qquad (\texttt{IRUN})\quad \frac{\mathcal{H} \vdash \underline{e} \rightarrowtail e}{\mathcal{H} \vdash \texttt{let}\ H = \underline{e}\ \texttt{in}\ (H\ \texttt{()}) \rightarrowtail \texttt{run}\ e}$$

Continuation Cumulation Operator

$$\overline{\bot} = \varnothing \qquad \overline{(K, \kappa)} = \overline{K} \cup \overline{\kappa}$$
$$\overline{[\cdot]} = \varnothing \qquad \overline{(\delta H.\kappa) \odot_\nu \underline{e}} = \overline{\kappa} \cup \{H : \underline{e}\}$$

Fig. 4. Inverse Translation from $\lambda_{\mathcal{C}}$ to $\lambda_{\mathcal{S}}$

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 $\lambda_{\mathcal{S}}$, 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:*

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

We write $e \xrightarrow{\mathcal{A}^*} e'$ for the reflexive, transitive closure of $\xrightarrow{\mathcal{A}}$. We write $e \xrightarrow{\mathcal{C};\mathcal{A}^*} e'$ for the reduction $\xrightarrow{\mathcal{C}}$ followed by zero or more reductions $\xrightarrow{\mathcal{A}}$ until no more reduction $\xrightarrow{\mathcal{A}}$ 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 $e, e' \in \textit{Expr}_{\mathcal{S}}$ and $e \xrightarrow{0} e'$. Let $\varnothing \vdash e \mapsto (\underline{e}, \bot)$ and $\varnothing \vdash e' \mapsto (\underline{e'}, \bot)$. Then $\underline{e} \xrightarrow{\mathcal{C};\mathcal{A}^*} \underline{e'}$. Furthermore, If $e \in \textit{Value}^0_{\mathcal{S}}$ then $\underline{e} \in \textit{Value}_{\mathcal{C}}$.*

3.3 Inverse Translation

Figure 4 presents an inverse translation from $\lambda_{\mathcal{C}}$ to $\lambda_{\mathcal{S}}$. We almost identically adopt the inverse translation of Choi et al. [4] The inverse translation judgment $\mathcal{H} \vdash \underline{e} \rightarrowtail e$ means that a $\lambda_{\mathcal{C}}$ expression $\underline{e}$

is translated back to a $\lambda_{\mathcal{S}}$ expression e under *hole environment* $\mathcal{H}$. 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 $\lambda_{\mathcal{S}}$ expression to $\lambda_{\mathcal{C}}$ expression, and then inversed it back to $\lambda_{\mathcal{S}}$, then the result is the same with the input. Together with Theorem 1, Theorem 2 means that $\lambda_{\mathcal{S}}$ is simulated by $\lambda_{\mathcal{C}}$.

$$e \xrightarrow{0} e' \implies \begin{array}{ccc} e & & e' \\ \downarrow & & \uparrow \\ \underline{e} & \xrightarrow{\mathcal{C};\mathcal{A}^*} & \underline{e'} \end{array}$$

Theorem 2 (Inverse Translation). *Let e be a $\lambda_{\mathcal{S}}$ expression and R be an environment stack. If $R \vdash e \mapsto (\underline{e}, K)$ then $\overline{K} \vdash \underline{e} \rightarrowtail e$.*

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 $\underline{e}$ be, respectively, a staged program and its translated unstaged version. If $[\![e]\!] \sqsubseteq \pi[\![\underline{e}]\!]$ and $\alpha \circ \pi \circ \underline{\gamma} \sqsubseteq \hat{\pi}$ then $\alpha[\![e]\!] \sqsubseteq \hat{\pi}[\![\underline{e}]\!]$.*

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

1. Arsac, J.J.: Syntactic source to source transforms and program manipulation. Commun. ACM 22(1), 43–54 (Jan 1979)
2. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using asts, gensym, and reflection. In: Proceedings of the 2nd international conference on Generative programming and component engineering. pp. 57–76. GPCE '03, Springer-Verlag New York, Inc., New York, NY, USA (2003)
3. Choi, J., Kang, J., Park, D., Yi, K.: Unstaging translation from metaml-like multi-staged calculus to context calculus. Tech. Rep. ROSAEC-2012-015, ROSAEC Center, Seoul National University (Mar 2012), `http://rosaec.snu.ac.kr/publish/2012/techmemo/ROSAEC-2012-015.pdf`
4. Choi, W., Aktemur, B., Yi, K., Tatsuta, M.: Static analysis of multi-staged programs via unstaging translation. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 81–92. POPL '11, ACM, New York, NY, USA (2011)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977)
6. Danvy, O.: Type-directed partial evaluation. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 242–257. POPL '96, ACM, New York, NY, USA (1996)
7. Davies, R., Pfenning, F.: A modal analysis of staged computation. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 258–270. POPL '96, ACM, New York, NY, USA (1996)
8. Engler, D.R.: Vcode: a retargetable, extensible, very fast dynamic code generation system. In: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation. pp. 160–170. PLDI '96, ACM, New York, NY, USA (1996)
9. Graham, P.: On LISP: Advanced Techniques for Common LISP. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)
10. Hashimoto, M., Ohori, A.: A typed context calculus. Theor. Comput. Sci. 266(1-2), 249–272 (Sep 2001)
11. Heintze, N.: Set-based analysis of ml programs. In: Proceedings of the 1994 ACM conference on LISP and functional programming. pp. 306–317. LFP '94, ACM, New York, NY, USA (1994)
12. Heintze, N.C.: Set based program analysis. Ph.D. thesis, Pittsburgh, PA, USA (1992), uMI Order No. GAX93-22866
13. Inoue, J., Taha, W.: Reasoning about multi-stage programs. In: 2012 European Symposium on Programming. ESOP '12 (2012), to appear
14. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)
15. Kamin, S., Aktemur, B., Katelman, M.: Staging static analyses for program generation. In: Proceedings of the 5th international conference on Generative programming and component engineering. pp. 1–10. GPCE '06, ACM, New York, NY, USA (2006)
16. Kim, I.S., Yi, K., Calcagno, C.: A polymorphic modal type system for lisp-like multi-staged languages. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 257–268. POPL '06, ACM, New York, NY, USA (2006)
17. Lee, P., Leone, M.: Optimizing ml with run-time code generation. In: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation. pp. 137–148. PLDI '96, ACM, New York, NY, USA (1996)
18. Plotkin, G.: Call-by-name, call-by-value, and the λ-calculus. Theoretical Computer Science 1, 125–159 (1975)
19. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, M.F.: C and tcc: a language and compiler for dynamic code generation. ACM Trans. Program. Lang. Syst. 21(2), 324–369 (Mar 1999)
20. Sheard, T., Jones, S.P.: Template meta-programming for haskell. SIGPLAN Not. 37(12), 60–75 (Dec 2002)

21. Steele, Jr., G.L.: Common LISP: the language (2nd ed.). Digital Press, Newton, MA, USA (1990)
22. Taha, W., Nielsen, M.F.: Environment classifiers. In: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 26–37. POPL '03, ACM, New York, NY, USA (2003)
23. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 203–217. PEPM '97, ACM, New York, NY, USA (1997)
24. Taha, W.M.: Multistage programming: its theory and applications. Ph.D. thesis (1999), aAI9949870