Unstaging Translation of Cross-Stage Persistent Multi-Staged Programs

Joonwon Choi Jeehoon Kang Daejun Park Kwangkeun Yi
Seoul National University
{jwchoi,jhkang,djpark,kwang}@ropas.snu.ac.kr

Abstract

We present a semantic-preserving unstaging translation of cross-stage persistent multi-staged programs into context calculus. 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. The 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. Coupled with Choi et al. [4], our unstaging translation provides static analysis framework for both of the two staging semantics: CSP and Lisp-like multi-staged languages.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Theory

Keywords Multi-staged languages, Context calculus, Static analysis, Unstaging translation, Semantics preservation, Abstract interpretation

1. Introduction

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

Static analysis of multi-staged programs, however, is challenging. Program code is no longer statically fixed, but rather dynamically constructed. For example, consider static analysis of the following two-staged program[1] (inspired by [4]):

```
$p := 0$
$s :=$ box $0$
while *cond* do
  $p := p + 2$
  $s :=$ box $(p +$ unbox $s)$
done
run $s$
```

Suppose that determining the number of iterations of the while loop is statically undecidable. In this case, we have to consider all possible values of p and s for every iteration. Particularly, s can have infinitely many code values as follows:

$$\{\texttt{box}\ 0, \texttt{box}\ (2+0), \texttt{box}\ (4+2+0), \cdots\}$$

The problem is how to statically analyze "run s" expression. A naive approach, enumerating infinitely many cases, is not realizable. We need a certain sophisticated approach to cope with this problem.

Recently, one possible solution is proposed: unstaging translation. Choi et al. [4] proposed static analysis of multi-staged programs via unstaging translation. They analyze 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.

Choi et al.'s approach itself, however, cannot be applied to cross-stage persistent (CSP) multi-staged programs [23]. With respect to variable scope, CSP and Lisp-like multi-staged languages have totally different semantics. For example, the program "$(\lambda x.\texttt{box}\ x)\ 0$" has different meaning in each language as follows (refer to Section 1.1 for more details):

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

Therefore, Choi et al.'s unstaging translation is not fit for CSP multi-staged programs. For example, consider the following unstaging translation of Choi et al.:

$$\begin{tikzcd}
(\lambda x.\texttt{box}\ x)\ 0 \arrow[r, mapsto] \arrow[d, "\texttt{CSP}"] & (\lambda x.\lambda\rho.\rho\cdot\texttt{x})\ 0 \arrow[d] \\
\texttt{box}\ 0 \arrow[r, phantom, "\not\mapsto"] & \lambda\rho.\rho\cdot\texttt{x}
\end{tikzcd}$$

The Choi et al.'s unstaged program (in the right) does not preserve the CSP semantics of the original program (in the left).

In this article, we present a semantics-preserving unstaging translation of CSP multi-staged programs. Our unstaging translation enables static analysis of CSP multi-staged programs in the

[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.

same way as Choi et al. [4]. 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. Coupled with Choi et al. [4], our unstaging translation provides static analysis framework for both of the two staging semantics: CSP and Lisp-like multi-staged languages.

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,

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

In CSP multi-staged language (in the left) x in $\mathtt{box}\ x$ is bound to λx, but in Lisp-like multi-staged language (in the right) x in $\mathtt{box}\ x$ is not bound to λx.

Also, CSP multi-staged language admits only capture-avoiding substitution. On the other hand, Lisp admits variable-capturing substitution. For example,

$$\begin{array}{ll} & (\lambda y.(\mathtt{box}\ (\lambda x.(\mathtt{unbox}\ y))))\ (\mathtt{box}\ x) \\ \xrightarrow{\mathtt{CSP}} & (\mathtt{box}\ (\lambda x'.(\mathtt{unbox}\ (\mathtt{box}\ x)))) \\ \xrightarrow{\mathtt{CSP}} & (\mathtt{box}\ (\lambda x'.x)) \end{array}$$

$$\begin{array}{ll} & (\lambda y.(\mathtt{box}\ (\lambda x.(\mathtt{unbox}\ y))))\ (\mathtt{box}\ x) \\ \xrightarrow{\mathtt{Lisp}} & (\mathtt{box}\ (\lambda x.(\mathtt{unbox}\ (\mathtt{box}\ x)))) \\ \xrightarrow{\mathtt{Lisp}} & (\mathtt{box}\ (\lambda x.x)) \end{array}$$

1.2 Unstaging Translation

A basic idea of our unstaging translation is as follows:

- We unstage `box` into lambda abstraction and `run` into lambda application with a dummy expression `()` as follows:[2]

  $$\begin{array}{rcl} \mathtt{box}\ 1 & \longmapsto & \lambda u.1 \\ \mathtt{run}\ (\mathtt{box}\ 1) & \longmapsto & (\lambda u.1)\ \mathtt{()} \end{array}$$

  Lambda abstraction and application behave in a similar way of `box` and `run`. Lambda abstraction is freezing until it meets lambda application, which is same with `box` that is freezing until it meets `run`.

- We pull out `unbox` outside the enclosing `box`. Suppose a `box` expression contains `unbox` inside. Translating the entire `box` expression into lambda abstraction makes the inner `unbox` to be frozen: the `unbox` cannot be evaluated any longer. This contradicts the semantics of `unbox`. For example, `unbox` expression is supposed to be evaluated inside the `box` as follows:[3]

  $$\mathtt{box}\ (\mathtt{unbox}\ (1+1)) \longrightarrow \mathtt{box}\ (\mathtt{unbox}\ 2)$$

  however, after translating `box` to lambda abstraction, the expression "$(1+1)$" is no longer able to be evaluated:

  $$\lambda u.(\mathtt{unbox}\ (1+1)) \not\longrightarrow \lambda u.(\mathtt{unbox}\ 2)$$

  Therefore, we need to pull out `unbox` outside the `box`, and make it to be put back inside after evaluation as follows:[4]

  $$\begin{tikzcd} \mathtt{box}\ (\cdots \mathtt{unbox}\ e \cdots) \arrow[d, bend left] \\ \longmapsto (\delta H.\ \lambda u.(\cdots \mathtt{unbox}\ H \cdots)) \odot e \end{tikzcd}$$

  For example, the previous example program should be translated as follows:

  $$\begin{tikzcd} \mathtt{box}\ (\mathtt{unbox}\ (1+1)) \arrow[r, mapsto] \arrow[dd] & (\delta H.\lambda u.(\mathtt{unbox}\ H)) \odot (1+1) \arrow[d] \\ & (\delta H.\lambda u.(\mathtt{unbox}\ H)) \odot 2 \arrow[d] \\ \mathtt{box}\ (\mathtt{unbox}\ 2) \arrow[r, mapsto] & \lambda u.(\mathtt{unbox}\ 2) \end{tikzcd}$$

- We use hole-filling application of context calculus [10] to pull out `unbox`. Hole-filling application is similar with lambda application, except for allowing variable-capturing substitution. For example, consider the following hole-filling application.

  $$(\delta H.\lambda x.H) \odot x \longrightarrow \lambda x.x$$

  Variable x is inserted into the hole H, and furthermore, becomes to be bound to λx. Note that this kind of substitution is not possible in the usual lambda application:

  $$(\lambda h.\lambda x.h)\ x \not\longrightarrow \lambda x.x$$

  This characteristic of hole-filling application is well fit for pulling out `unbox`, especially when `unbox` expression has free variables. For example, consider the following translation:

  $$\mathtt{box}\ (\lambda x.\mathtt{unbox}\ (\mathtt{box}\ x)) \longmapsto (\delta H.\lambda u.\ (\lambda x.\mathtt{unbox}\ H)) \odot (\lambda u.x)$$

  Originally bound to λx (in the left), x in $\mathtt{box}\ x$ becomes unbound after pulled out (in the right). However, hole-filling application makes the x to be re-bound to λx as follows:

  $$(\delta H.\lambda u.\ (\lambda x.\mathtt{unbox}\ H)) \odot (\lambda u.x) \longrightarrow \mathtt{box}\ (\lambda x.\mathtt{unbox}\ (\mathtt{box}\ x))$$

1.3 Comparisons

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

1.4 Organization

Section 2 defines both the CSP multi-staged calculus λ_S and the context calculus λ_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 λ_S

CSP multi-staged calculus λ_S is a call-by-value λ-calculus with staging constructs whose variables are cross-stage persistent.

2.1.1 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:

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

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.

[2] Throughout this paper, $\longmapsto$ means unstaging translation.

[3] For the simplicity, we use a somewhat wrong-typed program as an example.

[4] For the simplicity, we do not unstage `unbox`. Complete unstaging translation will be discussed in Section 3.

$$(\mathtt{APP}_{\mathcal{S}}) \quad \frac{e_1 \xrightarrow{n} e_1'}{e_1\ e_2 \xrightarrow{n} e_1'\ e_2} \qquad \frac{e \xrightarrow{n} e'}{v^n\ e \xrightarrow{n} v^n\ e'}$$

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

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

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

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

Variable Substitution

$$\begin{array}{ll}
[x \mapsto v]i & = i \\
[x \mapsto v]y & = \begin{cases} v & \text{if } x = y \\ y & \text{otherwise} \end{cases} \\
[x \mapsto v]\lambda y.e & = \begin{cases} \lambda y.e & \text{if } x = y \\ \lambda y'.[x \mapsto v]([y \mapsto y']e) & \text{otherwise, fresh } y' \end{cases} \\
[x \mapsto v](e_1\ e_2) & = [x \mapsto v]e_1\ [x \mapsto v]e_2 \\
 & \\
[x \mapsto v]\mathtt{box}\ e & = \mathtt{box}\ [x \mapsto v]e \\
[x \mapsto v]\mathtt{unbox}\ e & = \mathtt{unbox}\ [x \mapsto v]e \\
[x \mapsto v]\mathtt{run}\ e & = \mathtt{run}\ [x \mapsto v]e
\end{array}$$

Figure 1. Operational Semantics of Cross-Stage Persistent Multi-Staged Calculus $\lambda_{\mathcal{S}}$

2.1.2 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:

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

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

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

Atomic reductions are carried out only in $(\mathtt{APP}_{\mathcal{S}})_3$, $(\mathtt{UNB}_{\mathcal{S}})_2$ and $(\mathtt{RUN}_{\mathcal{S}})_2$. Both $(\mathtt{APP}_{\mathcal{S}})_3$ and $(\mathtt{RUN}_{\mathcal{S}})_2$ are applied at stage 0, but $(\mathtt{UNB}_{\mathcal{S}})_2$ is applied at stage 1. Rule $(\mathtt{RUN}_{\mathcal{S}})_2$ executes the code template. Rule $(\mathtt{UNB}_{\mathcal{S}})_2$ is similar to $(\mathtt{RUN}_{\mathcal{S}})_2$, but $\mathtt{unbox}$ expression is replaced with the code template v^1 and immediately frozen. The substitution is straightforward extension of that of lambda calculus.

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 not only variable substitution for usual lambda abstractions but also hole-filling substitution for hole-filling abstractions as follows:

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

Conceptually, H in the right is a first-class context bound by a hole-filling abstraction δH. The context H is replaced for x by hole-filling application $\odot$.

$$(\mathtt{APP}_{\mathcal{C}}) \quad \frac{e_1 \xrightarrow{\mathcal{C}} e_1'}{e_1\ e_2 \xrightarrow{\mathcal{C}} e_1'\ e_2} \qquad \frac{e \xrightarrow{\mathcal{C}} e'}{v\ e \xrightarrow{\mathcal{C}} v\ e'}$$

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

$$(\mathtt{HAPP}_{\mathcal{C}}) \quad \frac{e_1 \xrightarrow{\mathcal{C}} e_1'}{e_1 \odot_\nu e_2 \xrightarrow{\mathcal{C}} e_1' \odot_\nu e_2} \qquad \frac{e \xrightarrow{\mathcal{C}} e'}{v \odot_\nu e \xrightarrow{\mathcal{C}} v \odot_\nu e'}$$

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

Variable Substitution

$$\begin{array}{ll}
\{v/x\}i & = i \\
\{v/x\}y & = \begin{cases} v & \text{if } x = y \\ y & \text{otherwise} \end{cases} \\
\{v/x\}\lambda y.e & = \begin{cases} \lambda y.e & \text{if } x = y \\ \lambda y'.\{v/x\}(\overline{\{y'/y\}}\ e) & \text{otherwise, fresh } y' \end{cases} \\
\{v/x\}(e_1\ e_2) & = \{v/x\}e_1\ \{v/x\}e_2 \\
 & \\
\{v/x\}X^\nu & = X^\nu \\
\{v/x\}\delta X.e & = \delta X.\{v/x\}e \\
\{v/x\}e_1 \odot_\nu e_2 & = \begin{cases} \{v/x\}e_1 \odot_\nu \{v/x\}e_2 & \text{if } x \notin \mathit{dom}(\nu) \\ \{v/x\}e_1 \odot_\nu e_2 & \text{if } x \in \mathit{dom}(\nu) \end{cases}
\end{array}$$

Hole Substitution

$$\begin{array}{ll}
i[v/X] & = i \\
x[v/X] & = x \\
(\lambda x.e)[v/X] & = \lambda x.e[v/X] \\
(e_1\ e_2)[v/X] & = e_1[v/X]\ e_2[v/X] \\
Y^\nu[v/X] & = \begin{cases} \overline{\nu}\ v & \text{if } X = Y \\ Y^\nu & \text{if } X \neq Y \end{cases} \\
(\delta Y.e)[v/X] & = \begin{cases} \delta Y.e & \text{if } X = Y \\ \delta Y.e[v/X] & \text{if } X \neq Y \text{ and } Y \notin \mathit{FH}(v) \end{cases} \\
(e_1 \odot_\nu e_2)[v/X] & = e_1[v/X] \odot_\nu e_2[v/X]
\end{array}$$

Renamer Application

$$\begin{array}{ll}
\overline{\nu}\ x & = \nu(x) \\
\overline{\nu}\ \lambda x.e & = \lambda x.\overline{\nu|_{\mathit{dom}(\nu)\setminus\{x\}}}\ e \\
\overline{\nu}\ (e_1\ e_2) & = (\overline{\nu}\ e_1)\ (\overline{\nu}\ e_2) \\
\overline{\nu}\ X^{\nu'} & = X^{\nu \circ \nu'} \\
\overline{\nu}\ \delta X.e & = \delta X.\overline{\nu}\ e \\
\overline{\nu}\ e_1 \odot_{\nu'} e_2 & = \overline{\nu}\ e_1 \odot_{\nu'} \overline{\nu|_{\mathit{dom}(\nu)\setminus \mathit{dom}(\nu')}}\ e_2
\end{array}$$

Figure 2. Operational Semantics of Context Calculus $\lambda_{\mathcal{C}}$ [10]

2.2.1 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. To qualify the variables they depend on, hole variables and hole-filling applications have renamer annotations. Renamer $\nu = \{y_1/x_1, \cdots, y_n/x_n\}$ is a partial function from *Var* to *Var*. We identify a renamer ν to the extended total function by letting $\nu(x) = x$ for all $x \notin \mathit{dom}(\nu)$. Refer to [10] for more details.

2.2.2 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:

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

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

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

the variable $\colorbox{lightgray}{$x$}$ is not substituted for y, since $\colorbox{lightgray}{$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,

$$\begin{aligned}(\delta X.\lambda x.X^{\{x/y\}}) \odot_{\{y/z\}} z &\longrightarrow \lambda x.x \\ (\delta X.\lambda x.X^{\{\quad\}}) \odot_{\{\quad\}} z &\longrightarrow \lambda x.z\end{aligned}$$

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 $\lambda_{\mathcal{S}}$ to the context calculus $\lambda_{\mathcal{C}}$. Basic idea of our translation was presented in Section 1.2.

3.1 Translation Rules

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

Environments An environment is a list of binding variables. Each variables are introduced for each lambda abstraction (`TABS`). For example, for the expression $\lambda x_k.\lambda x_{k-1}.\cdots\lambda x_0.e$, the environment of e is $\{x_k; x_{k-1}; \cdots; x_0\}$. An environment stack is a list (stack) of environments, each of which corresponds to an enclosing `box`. For example, for given the following expression,

$$\lambda x_3.\texttt{box}\ (\lambda x_2.\lambda x_1.\texttt{box}\ (\lambda x_0.e))$$

the environment stack R of e is:

$$(\{x_3\}, \{x_2; x_1\}, \{x_0\})$$

Environments are used for creating renamer for `unbox` translation (`TUNB`). We will discuss more details in a later.

Continuations A continuation is a context with hole in the form of the following:

$$(\delta H_1.(\delta H_2.(\delta H_3.\cdots[\cdot]\cdots) \odot_{\nu_3} e_3) \odot_{\nu_2} e_2) \odot_{\nu_1} e_1$$

A continuation stack is a list (stack) of continuations. A continuation is generated for each `unbox` translation, and consumed later for each `box` translation.

Continuations are used for pulling out `unbox`. We will discuss more details in a later.

Definitions

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

Translation

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

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

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

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

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

$$\frac{R, \bot \vdash e \mapsto (\underline{e}, \bot) \qquad \texttt{new}\ u}{R \vdash \texttt{box}\ e \mapsto (\lambda u.\underline{e}, \bot)}$$

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

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

Renamer Inverse

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

Continuation Stack Merge

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

Figure 3. Translation from $\lambda_{\mathcal{S}}$ to $\lambda_{\mathcal{C}}$

Translation of variables As opposed to Choi et al. [4], a variable remains as it is during translation, in order to respect 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,

$$\begin{tikzcd} (\lambda x.\texttt{box}\ x)\ 0 \arrow[r, mapsto] \arrow[d] & (\lambda x.\lambda u.x)\ 0 \arrow[d] \\ \texttt{box}\ 0 \arrow[r, mapsto] & \lambda u.0 \end{tikzcd}$$

the constant 0 is replaced with the variable x after lambda application. Note that this is not the case in Lisp-like multi-staged language. Refer to Section 1.1 for more details.

Translation of boxes As we explained in Section 1.2, "`box` e" is translated into lambda abstraction "$\lambda u.\underline{e}$". In the meantime, `unbox` inside the e is pulled out, and put aside at the continuation κ. Later on, rule $(\texttt{TBOX})_1$ harvests the continuation to complete the translation of `box`. For example, consider the following translation:

$$\cfrac{\cfrac{\cfrac{\bot, \bot \vdash 1 \mapsto (1, \bot)}{\bot \vdash \texttt{box}\ 1 \mapsto (\lambda u.1, \bot)}}{\bot \vdash \texttt{unbox}\ (\colorbox{lightgray}{$\texttt{box}\ 1$}) \mapsto (H^{\{\}}\ \texttt{()}, \colorbox{lightgray}{$(\delta H.\boxed{[\cdot]}) \odot_{\{\}} \lambda u.1$})}}{\bot \vdash \texttt{box}\ (\texttt{unbox}\ (\colorbox{lightgray}{$\texttt{box}\ 1$})) \mapsto (\colorbox{lightgray}{$(\delta H.\boxed{\lambda u'.(H^{\{\}}\ \texttt{()})}) \odot_{\{\}} \lambda u.1$}, \bot)}$$

The expression "`box 1`" (shaded area in the left) is pulled out and temporarily saved at the continuation (shaded area in the upper-right). Then the continuation is consumed and merged with remaining expression (in the lower-right).

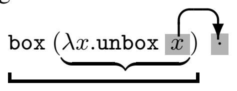
If box has no unbox inside, then continuation becomes empty and rule $(\mathtt{TBOX})_2$ simply finishes the translation. For example, refer to the second line in the above example.

Translation of unboxes As we explained in Section 1.2, "unbox e" is pulled outside. More specifically, the unbox expression e is pulled out and replaced with a hole variable H. In the meantime, rule (TUNB) generates a continuation using the expression e:

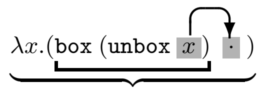
$$(\delta H.[\cdot]) \odot_\nu \underline{e}$$

This continuation is supposed to be consumed by $(\mathtt{TBOX})_1$. The hole "$[\cdot]$" is a placeholder in which enclosing box expression is to be inserted by the rule $(\mathtt{TBOX})_1$. For example, refer to the translation example in the previous paragraph.

Determining renamers In the course of generating the continuation, we need to carefully determine the renamer ν. As for the above example, the renamer ν should specify all free variables of "$\underline{e}$" that had been originally bound before being pulled out. For example, for the following program:[5]



the variable x should be specified by renamer. This is because the variable x (shaded) has been originally bound to λx, but it becomes unbound after being pulled out. On the other hand, for the following program:



nothing has to be specified. The variable x (shaded) is still bound after being pulled outside the box. Note that the spot "$\cdot$" also sits inside the scope of lambda binding.

All variables to be specified by renamer are represented in environments. Thus, rule (TUNB) consults with environment r_n for determining renamer ν. During translation, environments are elaborately manipulated for the purpose of identifying which variables to be specified by renamers. A comprehensive example is presented in Figure 4.

Translation of runs Conceptually, run expression "run e" is unstaged into lambda application "$\underline{e}$ ()". However, we unstage it into "let $h = \underline{e}$ in h ()", a syntactic sugar of "$\underline{e}$ ()". This is because we need to distinguish translated images of run from that of unbox. This syntactic information helps for inverse translation to find its original form easily. Refer to Section 3.3 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. Without loss of generality, we assume that all bound variables in $\lambda_{\mathcal{S}}$ are distinct.

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:

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

[5] For the following two examples, the under-brace ($\underbrace{\cdot}$) indicates the scope of lambda binding, and the under-bracket ($\underline{\cdot}$) indicates the scope of box expression. For the simplicity, we use a somewhat wrong-typed program as an example.

Definitions

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

Translation

$$(\mathtt{ICON}) \qquad \mathcal{H} \vdash i \rightarrowtail i$$

$$(\mathtt{IVAR}) \qquad \mathcal{H} \vdash x \rightarrowtail x$$

$$(\mathtt{IABS}) \qquad \frac{\mathcal{H} \vdash \underline{e} \rightarrowtail e}{\mathcal{H} \vdash \lambda x.\underline{e} \rightarrowtail \lambda x.e}$$

$$(\mathtt{IAPP}) \qquad \frac{\mathcal{H} \vdash \underline{e_i} \rightarrowtail e_i \quad \underline{e_2} \neq ()}{\mathcal{H} \vdash \underline{e_1}\ \underline{e_2} \rightarrowtail e_1\ e_2}$$

$$(\mathtt{ICTX}) \qquad \frac{\mathcal{H} \cup \{H : \underline{e'}\} \vdash \underline{e} \rightarrowtail e}{\mathcal{H} \vdash (\delta H.\underline{e}) \odot_\nu \underline{e'} \rightarrowtail e}$$

$$(\mathtt{IBOX}) \qquad \frac{\mathcal{H} \vdash \underline{e} \rightarrowtail e}{\mathcal{H} \vdash \lambda u.\underline{e} \rightarrowtail \mathtt{box}\ e}$$

$$(\mathtt{IUNB}) \qquad \frac{\mathcal{H} \vdash \mathcal{H}(H) \rightarrowtail e}{\mathcal{H} \vdash H\ () \rightarrowtail \mathtt{unbox}\ e}$$

$$(\mathtt{IRUN}) \qquad \frac{\mathcal{H} \vdash \underline{e} \rightarrowtail e}{\mathcal{H} \vdash \mathtt{let}\ H = \underline{e}\ \mathtt{in}\ (H\ ()) \rightarrowtail \mathtt{run}\ e}$$

Continuation Cumulation Operator

$$\begin{array}{llll} \overline{\perp} & = \varnothing & \overline{(K, \kappa)} & = \overline{K} \cup \overline{\kappa} \\ \overline{[\cdot]} & = \varnothing & \overline{(\delta H.\kappa) \odot_\nu \underline{e}} & = \overline{\kappa} \cup \{H : \underline{e}\} \end{array}$$

Figure 5. Inverse Translation from $\lambda_{\mathcal{C}}$ to $\lambda_{\mathcal{S}}$

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. Refer to [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 $\varnothing \vdash e \mapsto (\underline{e}, \perp)$ and $\varnothing \vdash e' \mapsto (\underline{e'}, \perp)$. 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 5 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 sub-expressions. Rule (IUNB) uses the hole environment, while rule (ICTX) stores dragged unbox sub-expression 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 $\lambda_{\mathcal{C}}$.

$$\begin{tikzcd} & e \arrow[d] & e' \\ e \arrow[r, "0"] & \underline{e} \arrow[r, "\mathcal{C};\mathcal{A}^*"] & \underline{e'} \arrow[u] \end{tikzcd} \quad \Longrightarrow$$

$$
\begin{array}{c}
\{x_1;x_2\} \vdash x_1 \mapsto x_1, \bot \qquad \{x_1;x_2\} \vdash x_2 \mapsto x_2, \bot \\
\hline
\{x_1;x_2\} \vdash x_1\ x_2 \mapsto x_1\ x_2, \quad \bot \\
\hline
\bot, \{x_1;x_2\} \vdash \texttt{unbox}\ (x_1\ x_2) \mapsto H_1^{\nu_1^{-1}}\ \texttt{()}, \quad (\delta H_1.[\cdot]) \odot_{\nu_1} (x_1\ x_2) \qquad \nu_1 = \{w_1/x_1, w_2/x_2\} \\
\hline
\bot, \{x_1\}, \{x_2\} \vdash \texttt{unbox}\ (\texttt{unbox}\ (x_1\ x_2)) \mapsto H_2^{\nu_2^{-1}}\ \texttt{()}, \quad (\delta H_1.[\cdot]) \odot_{\nu_1} (x_1\ x_2), \quad (\delta H_2.[\cdot]) \odot_{\nu_2} (H_1^{\nu_1^{-1}}\ \texttt{()}) \qquad \nu_2 = \{w_2/x_2\} \\
\hline
\bot, \{x_1\}, \bot \vdash \lambda x_2.(\texttt{unbox}\ (\texttt{unbox}\ (x_1\ x_2))) \mapsto \lambda x_2.(H_2^{\nu_2^{-1}}\ \texttt{()}), \quad (\delta H_1.[\cdot]) \odot_{\nu_1} (x_1\ x_2), \quad (\delta H_2.[\cdot]) \odot_{\nu_2} (H_1^{\nu_1^{-1}}\ \texttt{()}) \\
\hline
\bot, \{x_1\} \vdash \texttt{box}\ (\lambda x_2.(\texttt{unbox}\ (\texttt{unbox}\ (x_1\ x_2)))) \mapsto (\delta H_2.\lambda u_2.\lambda x_2.(H_2^{\nu_2^{-1}}\ \texttt{()})) \odot_{\nu_2} (H_1^{\nu_1^{-1}}\ \texttt{()}), \quad (\delta H_1.[\cdot]) \odot_{\nu_1} (x_1\ x_2) \\
\hline
\bot, \bot \vdash \lambda x_1.(\texttt{box}\ (\lambda x_2.(\texttt{unbox}\ (\texttt{unbox}\ (x_1\ x_2))))) \mapsto \lambda x_1.((\delta H_2.\lambda u_2.\lambda x_2.(H_2^{\nu_2^{-1}}\ \texttt{()})) \odot_{\nu_2} (H_1^{\nu_1^{-1}}\ \texttt{()})), \quad (\delta H_1.[\cdot]) \odot_{\nu_1} (x_1\ x_2) \\
\hline
\bot \vdash \texttt{box}\ (\lambda x_1.(\texttt{box}\ (\lambda x_2.(\texttt{unbox}\ (\texttt{unbox}\ (x_1\ x_2)))))) \mapsto (\delta H_1.\lambda u_1.\lambda x_1.((\delta H_2.\lambda u_2.\lambda x_2.(H_2^{\nu_2^{-1}}\ \texttt{()})) \odot_{\nu_2} (H_1^{\nu_1^{-1}}\ \texttt{()}))) \odot_{\nu_1} (x_1\ x_2), \quad \bot
\end{array}
$$

Figure 4. Example of unstaging translation: Environments are elaborately manipulated for the purpose of identifying which variables to be specified by renamers. Shaded areas demonstrate how renamers are determined.

Theorem 2 (Inverse Translation). *Let e be a λ_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. Application to Program Analysis

We analyze CSP multi-staged programs 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.

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.

For example, consider the following staged program e.[6]

```
p := 0
s := box 0                      (* indexed as u_1 *)
while cond do
  p := p + 2
  s := box (p + unbox s)        (* indexed as u_2 *)
done
run s
```

During the evaluation, s was originally assigned to $\texttt{box}\ 0$. After iterations, s becomes $\texttt{box}\ 2+0$ and then $\texttt{box}\ 4+2+0$. 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.

4.1 Unstaging Translation

To analyze the staged program e, we first unstage e into $\underline{e}$.

```
p := 0
s := λu_1.0
while cond do
  p := p + 2
  s := δH.λu_2.(p + H ()) ⊙ (s);
done
s ()
```

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 sub-expression is captured by the lambda abstraction at the left sub-expression, which is not a behavior of lambda abstraction and application (refer to Section 2.2).

[6] Note that this program exposes the characteristics of CSP multi-staged language: the variable p is defined at stage 0 and used at stage 1.

4.2 Static Analysis of the Context Calculus

For example, we present a set-based analysis [11, 12] on the context calculus. 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. Refer to [3] for more details.

We present the collecting semantics $[\![e]\!]$ and $[\![\underline{e}]\!]$ as concrete semantics.

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

p	has	$0, 2, 4, \cdots$
s	has	$\texttt{box}\ 0, \texttt{box}\ (2+0), \texttt{box}\ (4+2+0), \cdots$
$\texttt{run}\ s$	has	$0, 2, 6, \cdots$

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

p	has	$\langle 0, \varnothing\rangle, \langle 2, \varnothing\rangle, \langle 4, \varnothing\rangle, \cdots$
s	has	$\langle \lambda u_1.0, \varnothing\rangle,$
		$\langle \lambda u_2.(p_1 + H\ ()), \{p_1 \mapsto 2, H \mapsto \lambda u_1.0\}\rangle,$
		$\langle \lambda u_2.(p_2 + (\lambda u_2.(p_1 + H\ ()))\ ()),$
		$\{p_2 \mapsto 4, p_1 \mapsto 2, H \mapsto \lambda u_1.0\}\rangle,$
		$\cdots$
u_1	has	$\langle (), \varnothing\rangle$
u_2	has	$\langle (), \varnothing\rangle$
H	has	$\langle \lambda u_1.0, \varnothing\rangle,$
		$\langle \lambda u_2.(p_1 + H\ ()), \{p_1 \mapsto 2, H \mapsto \lambda u_1.0\}\rangle,$
		$\langle \lambda u_2.(p_2 + (\lambda u_2.(p_1 + H\ ()))\ ()),$
		$\{p_2 \mapsto 4, p_1 \mapsto 2, H \mapsto \lambda u_1.0\}\rangle,$
		$\cdots$
$s\ ()$	has	$\langle 0, \varnothing\rangle, \langle 2, \varnothing\rangle, \langle 6, \varnothing\rangle, \cdots$

Now we consider the abstract semantics $[\![\hat{e}]\!]$ and $[\![\hat{\underline{e}}]\!]$.

- $[\![\hat{e}]\!]$: 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 is going on in analysis.

Any static analysis can be employed in our framework. For the translated program, we get following result:

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

4.3 Projection

Finally, we present concrete projection π and abstract projection $\hat{\pi}$.

- 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.
- $\hat{\pi}$: 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:

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

$$\xmapsto{\hat{\pi}} \left\{ \begin{array}{rcl} p & \text{has} & 0, 2, 4, \cdots \\ s & \text{has} & \left\{ \begin{array}{l} S_2 \to u_2(S_3) \\ S_3 \to u_1 \mid u_2(S_3) \end{array} \right. \\ \texttt{run}\ s & \text{has} & \textit{values generated by } (V \to 0 \mid p + V) \end{array} \right.$$

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

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 $\alpha[\![e]\!] \sqsubseteq \hat{\pi}\widehat{[\![\underline{e}]\!]}$.*

5. Related Work

5.1 Translation

For comparisons to Choi et al. [4] and Inoue and Taha [13], refer to Section 1.3. 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.

5.2 Analysis

In terms of 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 for CSP multi-staged calculus and prove its correctness. 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. Coupled with Choi et al. [4], our unstaging translation provides static analysis framework for both of the two staging semantics: CSP and Lisp-like multi-staged languages.

References

[1] J. J. Arsac. Syntactic source to source transforms and program manipulation. *Commun. ACM*, 22(1):43–54, Jan. 1979. ISSN 0001-0782.

[2] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[3] J. Choi, J. Kang, D. Park, and K. Yi. Unstaging translation from metaml-like multi-staged calculus to context calculus. Technical Report ROSAEC-2012-015, ROSAEC Center, Seoul National University, Mar. 2012. `http://rosaec.snu.ac.kr/publish/2012/techmemo/ROSAEC-2012-015.pdf`.

[4] W. Choi, B. Aktemur, K. Yi, and M. Tatsuta. Static analysis of multi-staged programs via unstaging translation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 81–92, New York, NY, USA, 2011. ACM.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[6] O. Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 242–257, New York, NY, USA, 1996. ACM.

[7] R. Davies and F. Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 258–270, New York, NY, USA, 1996. ACM.

[8] D. R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 160–170, New York, NY, USA, 1996. ACM.

[9] P. Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[10] M. Hashimoto and A. Ohori. A typed context calculus. *Theor. Comput. Sci.*, 266(1-2):249–272, Sept. 2001. ISSN 0304-3975.

[11] N. Heintze. Set-based analysis of ml programs. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, LFP '94, pages 306–317, New York, NY, USA, 1994. ACM.

[12] N. C. Heintze. *Set based program analysis*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22866.

[13] J. Inoue and W. Taha. Reasoning about multi-stage programs. In *Proceedings of the 21st European conference on Programming Languages and Systems*, ESOP'12, pages 357–376, Berlin, Heidelberg, 2012. Springer-Verlag.

[14] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[15] S. Kamin, B. Aktemur, and M. Katelman. Staging static analyses for program generation. In *Proceedings of the 5th international confer-*

ence on Generative programming and component engineering, GPCE '06, pages 1–10, New York, NY, USA, 2006. ACM.

[16] I.-S. Kim, K. Yi, and C. Calcagno. 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*, POPL '06, pages 257–268, New York, NY, USA, 2006. ACM.

[17] P. Lee and M. Leone. Optimizing ml with run-time code generation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 137–148, New York, NY, USA, 1996. ACM.

[18] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[19] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Trans. Program. Lang. Syst.*, 21(2):324–369, Mar. 1999. ISSN 0164-0925.

[20] T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002. ISSN 0362-1340.

[21] G. L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[22] W. Taha and M. F. Nielsen. Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 26–37, New York, NY, USA, 2003. ACM.

[23] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '97, pages 203–217, New York, NY, USA, 1997. ACM.

[24] W. M. Taha. *Multistage programming: its theory and applications*. PhD thesis, 1999. AAI9949870.