Static Analysis of Multi-Staged Programs via Unstaging Translation

Wontae Choi\textsuperscript{1} \hspace{1cm} Baris Aktemur\textsuperscript{2} \hspace{1cm} Kwangkeun Yi\textsuperscript{1} \hspace{1cm} Makoto Tatsuta\textsuperscript{3}

\textsuperscript{1} Seoul National University, Korea \hspace{1cm} \textsuperscript{2} Ozyegin University/UIUC, Turkey \hspace{1cm} \textsuperscript{3} National Institute of Informatics, Japan

POPL 2011 @ Austin, USA
Multi-Staged Programming

Program codes are first class objects
“meta programming”
Multi-Staged Programming

A general concept that subsumes

- C++ and Haskell templates
- web programming’s runtime code generation
- macro
- Lisp’s quasi-quotation
- partial evaluation
Multi-Staged Programming

Divides a computation into stages

- stage 0 program: conventional program
- stage n+1 program: code value at stage n
Multi-Staged Programming

In presentation, we are going to use Lisp-like syntax + 2 stages

\[
e := \ldots
\]

- `e` code as a data
- `,e` code composition
- `run e` code execution
Multi-Staged Programming Examples

• code as a value
  \((1+1)\)

• open code
  \((x+1)\)

• code composition and intentional variable capturing
  \[
  \text{let } y = \text{\( (x+1) \)} \text{ in } \text{\( (\lambda x. ,y) \)} \rightarrow \text{\( (\lambda x. x+1) \)}
  \]

• code execution
  run \((1+1)\)
Contents

• Problem in Static Analysis
• Translation
• Projection
• Conclusion
Problem in Static Analysis

- Program text to analyze is dynamic
- Conventional analysis may fail to handle “run”

let spow n = if (n=0) then ‘1 else ‘(x* ,(spow (n-1)))
in let pow = ‘(λx. ,(spow input))
in (run pow) 2
Problem in Static Analysis

- Program text to analyze is dynamic
- Conventional analysis may fail to handle “run”

\[
\{1, (x*1), (x*x*1), \ldots\}
\]

```plaintext
let spow n = if (n=0) then 1 else (x, (spow (n-1)))
in let pow = (λx. ,(spow input))
in (run pow) 2
```
Problem in Static Analysis

- Program text to analyze is dynamic
- Conventional analysis may fail to handle “run”

```
let spow n = if (n=0) then '1' else ('x*' , (spow (n-1)))
in let pow = 'x' , (spow input))
in (run pow) 2
```
Problem in Static Analysis

- Program text to analyze is dynamic
- Conventional analysis may fail to handle “run”

\[
\text{let spow } n = \begin{cases} 
1 & \text{if (} n = 0 \text{)} \\
(x \times (\text{spow } (n - 1))) & \text{else}
\end{cases}
\]

\[
\text{in let pow = } (\lambda x. \times (\text{spow } \text{input}))
\]

\[
\text{in } (\text{run pow}) 2
\]
Problem in Static Analysis

- Program text to analyze is dynamic
- Conventional analysis may fail to handle “run”

```ml
let spow n = if (n=0) then 1 else (x * (spow (n-1)))
in let pow = (λx. (spow input))
in (run pow) 2
```

\{λx.1, λx.x*1, λx.x*x*1, ....\}

Static estimation: \$S \rightarrow 1 \mid x*S\$

Concretization: \{‘1, ‘(x*1), ‘(x*x*1), ...\}
Problem in Static Analysis

- Program text to analyze is dynamic
- Conventional analysis may fail to handle “run”

```
let spow n = if (n=0) then '1 else ('(x*1), ('x*x*1), ...)
in let pow = '(
  λx. , (spow input))
in (run pow) 2

{λx.1, λx.x*1, λx.x*x*1, ...}
```

Static estimation:

\[ S \rightarrow 1 | x*S \]

Concretization:

\[ \{ '1, 'x*1, 'x*x*1, ... \} \]

Unrealizable!
Our Contribution

- An unstaging translation which preserves the semantics
- An analysis framework based on the translation

\[ e : \text{staged program} \]

\[ \text{translation} \]

\[ e : \text{conventional program} \]

\[ \text{conventional analysis} \]

\[ \text{projection} \]

\[ \hat{e} \]

\[ e \]

\[ \text{analysis result } [\hat{e}] \text{ for } e \]

\[ \text{analysis result } [e] \text{ for } e \]
Theorems

- **Simulation**

- **Inversion**

- **Sound Projection**
Languages

Source Staged Language $\lambda_S$

<table>
<thead>
<tr>
<th>$e := \lambda x . e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mid e$</td>
</tr>
<tr>
<td>$\mid x$</td>
</tr>
<tr>
<td>$\mid \text{run } e$</td>
</tr>
</tbody>
</table>

Target Unstaged Language $\lambda_R$

<table>
<thead>
<tr>
<th>$e := \lambda x . e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mid e$</td>
</tr>
<tr>
<td>$\mid x$</td>
</tr>
<tr>
<td>$\mid {}$</td>
</tr>
<tr>
<td>$\mid e{x=e}$</td>
</tr>
<tr>
<td>$\mid e.x$</td>
</tr>
</tbody>
</table>
Translation Ideas (1/2)

• code expression to function expression

\[
\text{'(1+1) } \mapsto \lambda \rho. 1+1
\]

• free variable to record lookup

\[
\text{'(x+1) } \mapsto \lambda \rho. (\rho.x)+1
\]

• variable capturing to record passing

\[
\text{'(\lambda x. , (\text{'(x+1) } ) ) } \mapsto \lambda \rho_1. \lambda x. ((\lambda \rho_2. (\rho_2.x)+1) (\rho_1\{x=x\}))
\]

• run expression to application expression

\[
\text{run '}(1+1) \mapsto (\lambda \rho. 1+1) \{\}
\]
Translation Ideas (2/2)

• to preserve the evaluation order
Simulation

evaluation + translation

⇒ translation + evaluation + admin reduction
Inversion

\[
e \xrightarrow{\text{eager evaluation}} e' \\
\]

evaluation

\[\Rightarrow \text{translation + evaluation + admin reduction + inversion}\]
Static Analysis Framework

\[
\begin{align*}
\mathcal{D} & \ni [e] \xrightarrow{\gamma} \hat{\mathcal{D}} \\
\pi & \quad \hat{\mathcal{D}} \ni \hat{[e]}
\end{align*}
\]

Implementation

\[
e \mapsto e \quad \hat{[e]} \quad \hat{\pi}
\]

Requirement

\[
\alpha[e] \subseteq \pi[\hat{e}]
\]
Static Analysis Framework

Implementation

\[ e \rightarrow e \hat{\pi} \]

Requirement

\[ \alpha[e] \subseteq \hat{\pi}[\hat{e}] \]

Theorem

\[ [e] \subseteq \pi[e] \]

\[ \alpha \circ \pi \circ \gamma \subseteq \hat{\pi} \]

\[ \rightarrow \alpha[e] \subseteq \hat{\pi}[\hat{e}] \]
Example : Value Analysis

Setting 1) collecting analysis \([e]\) for the \textbf{staged} program (uncomputable)

staged program

\[
\text{let} \\
x = '0' \quad \text{(* indexed as } \rho_1 \text{ *)} \\
\text{repeat} \\
x = '(', x + 2) \quad \text{(* indexed as } \rho_2 \text{ *)} \\
\text{until } ? \\
in \\
\text{run } x
\]

\[
x \quad \text{has} \quad \{ '0', '(0+2)', '(0+2+2)', \ldots \}
\]

\[
(\text{run } x) \quad \text{has} \quad \{ 0, 2, 4, 6, \ldots \}
\]
Example: Value Analysis

Setting 2) collecting analysis \([e]\) for its translated version (uncomputable)

translated program

```plaintext
let
  x = (\rho_1.0)
repeat
  x = ((\lambda h. \lambda \rho_2.(h \rho_2)+2) x)
until ?
in
  x {}
```

\(x, h\) has  \(\{\langle \lambda \rho_1.0, \emptyset \rangle, \langle \lambda \rho_2.(h \rho_2)+2, \{h\mapsto \langle \lambda \rho_1.0, \emptyset \rangle \rangle, \ldots\}\}\)

\(\rho_1, \rho_2\) has  \(\{\}\)

\((x \{\}\) has  \(\{0, 2, 4, 6, \ldots\}\)
Example : Value Analysis

Setting 3) collecting projection \( \hat{\pi} \) (uncomputable)

\[
\begin{align*}
\text{translation + collecting analysis (part of)} \quad & \quad \text{projection result} \\
& \\
\hat{\pi} \\
\text{x, h has} & \{ \langle \lambda \rho_1.0, \emptyset \rangle, \langle \lambda \rho_2.(h \rho_2)+2, \{h \mapsto \langle \lambda \rho_1.0, \emptyset \rangle \} \rangle, \\
& \text{..... } \} \\
\rho_1, \rho_2 \text{ has} & \{ \} \\
\text{projection result} & \xrightarrow{\pi} \text{x has} \{ \text{`0,` (0+2), } \\
& \text{.....} \} \\
\end{align*}
\]

• inverse translation + removing unnecessary stuff
• intuition : "\( \lambda \rho \)" \( \xrightarrow{\hat{\pi}} \) "code \( \rho \)"

\( h \rho \) \( \xrightarrow{\hat{\pi}} \) "code-filling by h"

• \( \pi \) satisfies \( \hat{\pi} \)’s first safety condition : \( [e] \subseteq \pi [e] \)
Example: Value Analysis

(computable) **static** analysis $\hat{[e]}$ for the translated version

**translated program**

```
let
  x = (\rho_1.0)
repeat
  x = ((h.\rho_2.(h \rho_2)+2) x)
until ?

in
  x {}
```

<table>
<thead>
<tr>
<th></th>
<th>has</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$\lambda \rho_1.0$</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>$\lambda \rho_2.(h \rho_2)+2$</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>$\lambda \rho_1.0$</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>$\lambda \rho_2.(h \rho_2)+2$</td>
<td></td>
</tr>
<tr>
<td>$\rho_1, \rho_2$</td>
<td>has</td>
<td>${}$</td>
</tr>
<tr>
<td>$(x {}$)</td>
<td>has</td>
<td>0</td>
</tr>
<tr>
<td>$(x {}$)</td>
<td>has</td>
<td>$(h \rho_2) + 2$</td>
</tr>
<tr>
<td>$(h \rho_2)$</td>
<td>has</td>
<td>0</td>
</tr>
<tr>
<td>$(h \rho_2)$</td>
<td>has</td>
<td>$(h \rho_2) + 2$</td>
</tr>
</tbody>
</table>

**set-constraint style 0-CFA**
Example: Value Analysis

(computable) **static** analysis $\hat{[e]}$ for the translated version

translated program

```
let
  x = (λρ₁.0)
repeat
  x = (((λh.λρ₂.(h ρ₂)+2) x)
until ?
in
  x {}
```

(x { })’s values in grammar : $V \rightarrow 0 \mid V+2$

V -> 0 | V+2
Example: Value Analysis

(computable) **abstract** projection

**static analysis for the translated program**

\[
\begin{align*}
\text{x} & \text{ has } \lambda \rho_1.0 \\
\text{x} & \text{ has } \lambda \rho_2. (h \rho_2) + 2 \\
h & \text{ has } \lambda \rho_1.0 \\
h & \text{ has } \lambda \rho_2. (h \rho_2) + 2 \\
(x \ \{\} ) & \text{ has } V \rightarrow 0 \mid V + 2
\end{align*}
\]

**abstract projection result**

\[
\begin{align*}
x & \text{ has } S_1 \rightarrow \rho_1 \\
x & \text{ has } S_2 \rightarrow \rho_2(S) \\
S & \rightarrow \rho_1 \mid \rho_2(S) \\
(\text{run } x) & \text{ has } V \rightarrow 0 \mid V + 2
\end{align*}
\]

• **intuition**: 
  \[
  \begin{align*}
  \lambda \rho & \xrightarrow{\hat{\pi}} \text{"code } \rho\text{"} \\
h \rho & \xrightarrow{\hat{\pi}} \text{"code-filling by } h\text{"}
\end{align*}
  \]

• \(\hat{\pi}\) satisfies the second safety condition: \(\alpha \circ \pi \circ \gamma \sqsubseteq \hat{\pi}\)
Example : Value Analysis

final result for the staged program

let
  x = '0 (* indexed as $\rho_1$ *)
repeat
  x = '(' , x+2 ) (* indexed as $\rho_2$ *)
until ?
in
run x

"translation + static analysis + projection" is sound

$\alpha[e] \subseteq \hat{\pi} [\hat{e}]$
Conclusion

• Semantics-preserving translation from staged programs to conventional programs

• Sound analysis framework using the translation
Conclusion

• Semantics-preserving translation from staged programs to conventional programs

• Sound analysis framework using the translation

Unstaging + Conventional static analysis
That’s sufficient!

Thank you