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-quototation
- 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 := ...  
| 'e       code as a data
| ,e       code composition
| run e    code execution
Multi-Staged Programming Examples

• code as a value

\[
\text{\small '(1+1)}
\]

• open code

\[
\text{\small '(x+1)}
\]

• code composition and intentional variable capturing

\[
\text{let } y = \text{'(x+1) in \text{'(\lambda x. ,y) \rightarrow \text{'(\lambda x. x+1)}}}
\]

• code execution

\[
\text{run \text{'(1+1)}}
\]
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”

```
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”

```haskell
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 = (λx. , (spow input))
in (run pow) 2
```
Problem in Static Analysis

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

\[
\begin{align*}
\text{let } \text{spow } n &= \text{if } (n=0) \text{ then } '1' \text{ else } '(x*1), (x*x*1), \ldots' \\
\text{in let } \text{pow} &= '((\lambda x. ,\text{spow input}))' \\
\text{in (run pow)} \ 2
\end{align*}
\]

\[
\begin{align*}
\{1, (x*1), (x*x*1), \ldots\} \\
\text{pow} &\rightarrow \lambda x. S \\
\{\lambda x.1, \lambda x.x*1, \lambda x.x*x*1, \ldots\}
\end{align*}
\]

\[
\begin{align*}
S &\rightarrow 1 \ | \ x*S \\
\text{static estimation}
\end{align*}
\]
Problem in Static Analysis

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

\[
\text{let spow } n = \text{if } (n=0) \text{ then } '1' \text{ else } 'x' \ast (\text{spow } (n-1)) \\\n\text{in let pow = } ' (\lambda x. , (\text{spow input}) ) \\\n\text{in (run pow) } 2
\]

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

\[
\text{S \rightarrow 1 | xS}
\]

\[
\text{pow \rightarrow } \lambda x. S
\]

\[
\text{Unrealizable!}
\]
Our Contribution

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

• Simulation

\[ \begin{align*}
    e & \rightarrow e' \\
    \downarrow & \quad \downarrow \\
    e & \quad e'
\end{align*} \implies \begin{align*}
    e & \rightarrow e' \\
    \downarrow & \\
    e & \rightarrow e'
\end{align*} \]

• Inversion

\[ \begin{align*}
    e & \rightarrow e' \quad \implies \quad e' \rightarrow e \\
    \downarrow & \quad \uparrow \\
    e & \quad e'
\end{align*} \]

• Sound Projection

\[ \begin{align*}
    e & \quad [e] \in D \\
    \downarrow & \\
    e & \quad [e]
\end{align*} \quad \begin{align*}
    \gamma & \alpha \\
    \hats & \hat{D} \equiv [\hat{e}]
\end{align*} \quad \begin{align*}
    \pi & \\
    \hats & \hat{\pi} \\
    \hats & \hat{D} \equiv [\hat{e}]
\end{align*} \]
Languages

Source Staged Language $\lambda_S$

\[ e := \lambda x . e \]
\[ | e \ e \]
\[ | x \]
\[ | 'e \]
\[ | ,e \]
\[ | \text{run } e \]

Target Unstaged Language $\lambda_R$

\[ e := \lambda x . e \]
\[ | e \ e \]
\[ | x \]
\[ | {} \]
\[ | e\{x=e\} \]
\[ | e . x \]
Translation Ideas (1/2)

• code expression to function expression
  \((1+1) \rightarrow \lambda \rho.1+1\)

• free variable to record lookup
  \((x+1) \rightarrow \lambda \rho.(\rho.x)+1\)

• variable capturing to record passing
  \((\lambda x.\,(x+1)) \rightarrow \lambda \rho_1.\lambda x.((\lambda \rho_2.(\rho_2.x)+1)\ (\rho_1\{x=x\}))\)

• run expression to application expression
  \(\text{run } (1+1) \rightarrow (\lambda \rho.1+1) \{\}\)
Translation Ideas (2/2)

- to preserve the evaluation order
Evaluation + translation

\Rightarrow \text{translation} + \text{evaluation} + \text{admin reduction}
Inversion

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

Implementation

\[ e \mapsto [e] \in D \overset{\gamma}{\longleftarrow} \hat{D} \]

Requirement

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

\[ e \xrightarrow{\alpha} \pi \xrightarrow{\gamma} \hat{\pi} \]

Implementation

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

Requirement

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

Theorem

\[ \left\{ \begin{array}{c}
[e] \subseteq \pi[e] \\
\alpha \circ \pi \circ \gamma \subseteq \hat{\pi}
\end{array} \right\} \implies \alpha[e] \subseteq \hat{\pi} \hat{\pi} \]
Example : Value Analysis

Setting 1) collecting analysis $[e]$ for the staged program (uncomputable)

staged program

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

$x$ has $\{0, (0+2), (0+2+2), \ldots\}$

(run $x$) has $\{0, 2, 4, 6, \ldots\}$
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 \( \pi \) (uncomputable)

- 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: \( [e] \subseteq \pi[e] \)
(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 {}
```

<table>
<thead>
<tr>
<th></th>
<th>has</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$\lambda ρ₁.0$</td>
</tr>
<tr>
<td>x</td>
<td>$\lambda ρ₂.(h ρ₂)+2$</td>
</tr>
<tr>
<td>h</td>
<td>$\lambda ρ₁.0$</td>
</tr>
<tr>
<td>h</td>
<td>$\lambda ρ₂.(h ρ₂)+2$</td>
</tr>
<tr>
<td>$ρ₁, ρ₂$</td>
<td>{}</td>
</tr>
<tr>
<td>(x {})</td>
<td>0</td>
</tr>
<tr>
<td>(x {})</td>
<td>$(h ρ₂) + 2$</td>
</tr>
<tr>
<td>(h $ρ₂$)</td>
<td>0</td>
</tr>
<tr>
<td>(h $ρ₂$)</td>
<td>$(h ρ₂) + 2$</td>
</tr>
</tbody>
</table>

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

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

translated program

```plaintext
let
  x = (λρ₁.0)
repeat
  x = (((λh.λρ₂.(h ρ₂)+2) x)
until ?
in
x {}'s values in grammar : $V \rightarrow 0 \mid V+2$
```

\[
\begin{align*}
  x & \text{ has } \lambda\rho_1.0 \\
  x & \text{ has } \lambda\rho_2.(h \rho_2)+2 \\
  h & \text{ has } \lambda\rho_1.0 \\
  h & \text{ has } \lambda\rho_2.(h \rho_2)+2 \\
  \rho_1, \rho_2 & \text{ has } \{} \\
  (x \{\}) & \text{ has } 0 \\
  (x \{\}) & \text{ has } (h \rho_2) + 2 \\
  (h \rho_2) & \text{ has } 0 \\
  (h \rho_2) & \text{ has } (h \rho_2) + 2
\end{align*}
\]
Example: Value Analysis

(computable) **abstract** projection

**Static analysis for the translated program**

- x has \( \lambda \rho_1.0 \)
- x has \( \lambda \rho_2.(h \rho_2)+2 \)
- h has \( \lambda \rho_1.0 \)
- h has \( \lambda \rho_2.(h \rho_2)+2 \)
- \((x \{\})\) has \( V \to 0 \mid V+2 \)

**Abstract projection result**

- x has \( S_1 \to \rho_1 \)
- x has \( S_2 \to \rho_2(S) \)
- S \to \rho_1 \mid \rho_2(S) \)
- \((\text{run } x)\) has \( V \to 0 \mid V+2 \)

- **Intuition:**
  - \( \lambda \rho \) \(\xrightarrow{\hat{\pi}}\) "code \( \rho \)"
  - \( h \rho \) \(\xrightarrow{\hat{\pi}}\) "code-filling by \( h \)"

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

final result for the staged program

```
let
  x = '0' (* indexed as ρ₁ *)
repeat
  x = '(' x, x+2 ) (* indexed as ρ₂ *)
until ?
in
run x
```

```
translation + static analysis + projection

x has S₁ -> ρ₁
x has S₂ -> ρ₂(S)
S -> ρ₁ ∣ ρ₂(S)
(run x) has V -> 0 ∣ V+2
```

“translation + static analysis + projection” is sound

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