Collage of Static Analysis

- 0.5hr: Static Analysis Overview
- 1.5hr: Static Analysis Design Framework
- 1.0hr: Static Analysis Engineering Framework
- 1.0hr: Static Analysis of Multi-Staged Programs
Static Analysis of
Multi-Staged Programs

Kwangkeun Yi

Seoul National University, Korea
http://ropas.snu.ac.kr/~kwang

2/26/2012 – 3/2/2012
17th Estonian Winter School in Computer Science, Palmse,
Estonia

(co-work with I. Kim, C. Calcagno, W. Choi, B. Aktemur, M. Tatsuda)
Outline

- Multi-Staged Programming (MSP)
- Special Static Analysis of MSP (POPL’06)
- General Static Analysis of MSP (POPL’11)
program texts (code) as first class objects
“meta programming”

A general concept that subsumes
- web program’s runtime code generation
- macros & templates
- Lisp’s quasi-quotation
- partial evaluation

Common in JavaScript, Perl, PHP, Python, Lisp/Scheme, C’s macros, C++ & Haskell’s templates, C#, etc.
Multi-Staged Programming (2/4)

- divides a computation into stages
- program at stage 0: conventional program
- program at stage $n + 1$: code as data at stage $n$

<table>
<thead>
<tr>
<th>Stage</th>
<th>Computation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>usual + code + run</td>
<td>usual + code</td>
</tr>
<tr>
<td>$&gt;0$</td>
<td>code substitution</td>
<td>code</td>
</tr>
</tbody>
</table>
In examples, we will use Lisp-style staging constructs + only 2 stages

\[ e ::= \ldots \]
\[ \text{'e} \quad \text{code as data} \]
\[ ,e \quad \text{code substitution} \]
\[ \text{run } e \quad \text{execute code} \]

- code as a value: \text{'(1+1)}
- code composition: \text{let y = '(x+1) in } '(\lambda x.,y)
- code execution: \text{run } '(1+1)
Specializer/Partial evaluator

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

v.s. \[ \text{power}(x, 3) = x \times x \times x \]

prepared as

let \( spower(n) = \begin{cases} '1' & \text{if } n = 0 \\ '(x*, (spower(n-1)))' & \text{else} \end{cases} \)

let \( \text{fastpower} = '(\lambda x., (spower input))' \)

in (run fastpower) 2
Practice of Multi-Staged Programming

- open code
  
  `'(x+1)

- intentional variable-capturing substitution

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

- capture-avoiding substitution

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

- imperative operations with open code

  \[ \text{cell} := '(x+1); \ldots \text{cell} := '(y 1); \]
A static type system that supports the practice.

- type safety and
- the expressiveness of fully-fledged multi-staging operators

Previous type systems support only part of the practice.
A general, static analysis method for multi-staged programs.

The objects (program texts) to analyze
- are dynamic entities, which
- are only estimated by static analysis

Conventional analysis may fail to handle “run e”

No general static analysis method before.
variable-capture allowed at stages > 0 (the practice of 30 yrs)

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

variable-capture disallowed + “cross-stage persistence” (language-theory orthodox)

\[
(\lambda x.\text{'}x) 1
\]
A type system for (ML + Lisp’s quasi-quote system) supports all in multi-staged programming practice:
- open code: ‘(x+1)
- unrestricted imperative operations with open code
- intentional var-capturing substitution at stages > 0
- capture-avoiding substitution at stages > 0
- conservative extension of ML’s **let-polymorphism**
- principal type inference algorithm

[Kim, Yi, Calcagno 2006] A Let-Polymorphic Modal Type System for Lisp-style Multi-Staged Programming
Ideas

- code’s type: parameterized by its expected context
  \[ \square (\Gamma \triangleright int) \]

- view the type environment \( \Gamma \) as a record type
  \[ \Gamma = \{ x : int, y : int \rightarrow int, \cdots \} \]

- stages by the stack of type environments (modal logic S4)
  \[ \Gamma_0 \cdots \Gamma_n \vdash e : A \]

- with “due” restrictions
  - let-polymorphism for syntactic values
  - monomorphic \( \Gamma \) in code type \( \square (\Gamma \triangleright int) \)
  - monomorphic store types

Natural ideas worked.
Simple Type System

**Type** \( A, B \ ::= \ i \ | \ A \rightarrow B \ | \ \square(\Gamma \triangleright A) \)

code type

\('(x+1)': \(\square(\{x: \text{int}, \cdots\} \triangleright \text{int})\)

typing judgment

\[ \Gamma_0 \cdots \Gamma_n \vdash e: A \]

(TSBOX)

\[ \frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e: A}{\Gamma_0 \cdots \Gamma_n \vdash \text{box } e: \square(\Gamma \triangleright A)} \]

(TSUNBOX)

\[ \frac{\Gamma_0 \cdots \Gamma_n \vdash e: \square(\Gamma_{n+k} \triangleright A)}{\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash \text{unbox}_k e: A} \]

(TSEVAL)

\[ \frac{\Gamma_0 \cdots \Gamma_n \vdash e: \square(\emptyset \triangleright A)}{\Gamma_0 \cdots \Gamma_n \vdash \text{run } e: A} \]  

(for alpha-equiv. at stage 0)
A combination of
- ML’s let-polymorphism
  - syntactic value restriction + multi-staged “expansive$^n(e)$”
  - expansive$^n(e) = False$
    \[ \implies e \text{ never expands the store during its eval. at } \forall \text{stages } \leq n \]
  - e.g.) ‘(λx., e) : can be expansive
  - ‘(λx.run y) : unexpansive

- Rémy’s record types [Rémy 1993]
  - type environments as record types with field addition
  - record subtyping + record polymorphism
• if $e$ then ‘(x+1) else ‘1: $\square (\{x: \text{int}\} \rho \triangleright \text{int})$
  
  • then-branch: $\square (\{x: \text{int}\} \rho' \triangleright \text{int})$
  • else-branch: $\square (\rho'' \triangleright \text{int})$

let $x = 'y$ in ‘(,x + w); ‘((,x 1) + z)

  $x: \forall \alpha \forall \rho. \square (\{y: \alpha\} \rho \triangleright \alpha)$

  • first $x$: $\square (\{y: \text{int}, w: \text{int}\} \rho' \triangleright \text{int})$
  • second $x$: $\square (\{y: \text{int} \rightarrow \text{int}, z: \text{int}\} \rho'' \triangleright \text{int} \rightarrow \text{int})$
Type Inference Algorithm

- **Unification:**
  - Rémy’s unification for record type $\Gamma$
  - usual unification for new type terms such as $\Box(\Gamma \triangleright A)$ and $A\text{ ref}$

- **Sound and complete principal type inference:**
  - the same structure as top-down version $\mathcal{M}$ [Lee and Yi 1998] of the $\mathcal{W}$
  - usual on-the-fly instantiation and unification
A general, static analysis method for multi-staged programs.

The objects (program texts) to analyze
- are dynamic entities, which
- are only estimated by static analysis

Conventional analysis may fail to handle “run e”
- how to analyze the run of estimated program texts?

[Choi, Aktemur, Yi, Tatsuda 2011] Static Analysis of Multi-Staged Programs via Unstaging Translation
The set of possible code for $x$:

$$\{ '0, ' (0+2), ' (0+2+2), \cdots \}.$$ 

must first be finitely approximated, e.g., by a grammar:

$$S \to 0 \mid S+2.$$ 

analyzing “run $x$” needs code, not the grammar.
Our Solution

a detour: translate, analyze, and project.

1. unstaging translation
   - proof of semantic-preserving

2. conventional static analysis
   - can apply all existing static analysis techniques

3. cast the result back in terms of original staged programs
   - a sound condition for the projection
   - i.e., to be aligned with the correspondence induced by the translation.
Translation Languages

<table>
<thead>
<tr>
<th>Staged source</th>
<th>Unstaged target</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e ::= \lambda x.e$</td>
<td>$e ::= \lambda x.e$</td>
</tr>
<tr>
<td>$e\ e$</td>
<td>$e\ e$</td>
</tr>
<tr>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td><code>'e</code></td>
<td><code>}</code></td>
</tr>
<tr>
<td><code>,e</code></td>
<td><code>e{\text{x}=e}</code></td>
</tr>
<tr>
<td><code>run e</code></td>
<td><code>e \cdot x</code></td>
</tr>
</tbody>
</table>
Translation Ideas (1/2)

- Code into env-taking function:
  \[ (1+1) \mapsto \lambda \rho.1+1 \]

- Free variable in a code into record lookup:
  \[ (x+1) \mapsto \lambda \rho.(\rho \cdot x) + 1 \]

- Run expression into an application:
  \[ \text{run } (1+1) \mapsto (\lambda \rho.1+1)\{\} \]
code composition into an app. whose actual param. is for the code-to-be-plugged expr.:

\[(\lambda h. (\lambda \rho. (h \, \rho) + 2)) \, y\]

variable capturing into record passing + lookup:

\[(\lambda x. ((x + 1))) \rightarrow \lambda \rho_1 \lambda x. ((\lambda \rho_2. (\rho_2 \cdot x) + 1) \, (\rho_1 \{x = x\}))\]
Translation Example

\[
x := '0;
\]
\[
\text{repeat}
\]
\[
x := '(,x + 2)
\]
\[
\text{until } \text{cond};
\]
\[
\text{run } x
\]

\[
x := \lambda \rho . 0;
\]
\[
\text{repeat}
\]
\[
x := (\lambda h . (\lambda \rho . (h \ \rho) + 2)) \ x
\]
\[
\text{until } \text{cond};
\]
\[
x \ \{}\}
\]
Theorem

(Simulation) Let $e$ be a stage-$n$ $\lambda_S$ expression with no free variables such that $e \xrightarrow{n} e'$. Let $R \vdash e \mapsto (e, K)$ and $R \vdash e' \mapsto (e', K')$. Then $K(e) \xrightarrow{R; A^*} K'(e')$. 

\[
\begin{array}{c}
\vdots
\end{array}
\]
Theorem

(Inversion) Let $e$ be a $\lambda_S$ expression and $R$ be an environment stack. If $R \vdash e \leftrightarrow (e, K)$, then $H \vdash e \leftrightarrow e$ for any $H$ such that $\overline{K} \subseteq H$.
**Theorem**

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

\[ e \rightarrow [e] \in D_S \xrightarrow{\gamma} \hat{D}_S \ni [\hat{e}] \]

\[ e \rightarrow [e] \in D_R \xrightarrow{\gamma} \hat{D}_R \ni [\hat{e}] \]
Example (1/5): $\llbracket e \rrbracket$ staged collecting semantics

\begin{verbatim}
x := '0;
repeat
    x := '(', x + 2)
until cond;
run x
\end{verbatim}

Collecting semantics $\llbracket e \rrbracket =$

\begin{align*}
x & \text{ has } \{ '0, ' (0+2), ' (0+2+2), \cdots \} \\
\text{run } x & \text{ has } \{ 0, 2, 4, 6, \cdots \}
\end{align*}
Example (2/5): \([e] \text{ unstaged collecting semantics}\)

\[
x := \lambda \rho_1.0;
\]

\[
\text{repeat}
\]
\[
x := (\lambda h.(\lambda \rho_2.(h \rho_2)+2)) x
\]

\[
\text{until cond;}
\]
\[
x \{}\]

Collecting semantics \([e] =\)

\[
\begin{align*}
x, h & \text{ has } \{\langle \lambda \rho_1.0, \emptyset \rangle, \langle \lambda \rho_2.(h \rho_2)+2, \{h \mapsto \langle \lambda \rho_1.0 \rangle \} \rangle, \cdots \} \\
\rho_1, \rho_2 & \text{ has } \{\} \\
x \{} & \text{ has } \{0, 2, 4, 6, \cdots \}
\end{align*}
\]
Collecting semantics are aligned:

\[ \llbracket e \rrbracket \subseteq \pi \llbracket e \rrbracket \]

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

\[ \rho_1, \rho_2 \text{ has } \{\} \]

- \( \pi = \) inverse translation + removing admin stuff
- intuition

\[ "\lambda \rho" \mapsto \pi \text{ "code indexed as } \rho" \]

\[ "h \rho" \mapsto \pi \text{ "code-filling by } h" \]
Example (4/5): \( \hat{e} \) unstaged conventional analysis

\[
x := \lambda \rho_1 . 0;
repeat
  x := (\lambda h . (\lambda \rho_2 . (h \: \rho_2) + 2)) \: x
until \text{cond};
\]

\( x \) has \( \lambda \rho_1 . 0 \)
\( x \) has \( \lambda \rho_2 . (h \: \rho_2) + 2 \)
\( h \) has \( \lambda \rho_1 . 0 \)
\( h \) has \( \lambda \rho_2 . (h \: \rho_2) + 2 \)

0-CFA analysis \( \hat{e} \) in set-constraint style

\( x \) has \( \lambda \rho_1 . 0 \) \( \rightarrow V_1 \rightarrow 0 \mid V_1 + 2 \)
\( x \) has \( \lambda \rho_2 . (h \: \rho_2) + 2 \) \( \rightarrow V_2 \rightarrow 0 \mid V_1 + 2 \)
Example (5/5): \( \hat{\pi} \) projection of analysis

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

- intuition

"\( \lambda \rho \)" \( \hat{\pi} \) "code indexed as \( \rho \)"

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

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

- and was \( [e] \subseteq \pi[e] \)

Hence, by the projection theorem, correct:

\[
\alpha[e] \subseteq \hat{\pi}[\hat{e}]
\]
- semantic-preserving unstaging translation
- sound static analysis framework using the translation

\[
\begin{align*}
e & \quad \gamma \quad \alpha \\
\alpha \quad \gamma \quad \pi \quad \hat{\pi}
\end{align*}
\]

\[
\begin{align*}
[e] & \in D_S \quad \hat{D}_S \ni \hat{[e]} \\
[e] & \in D_R \quad \hat{D}_R \ni \hat{[e]}
\end{align*}
\]

unstaging + usual static analysis + projection are sufficient.
Things to Do

- Extend the design (theory) to “string-based” (unstructured) multi-staged programming
- Realistic static analyses
  - e.g. static Javascript malware detection
- Language design for multi-staged Dalvik (for “evolving” apps)
Messages from My Lectures
Static Analysis Overview

Static Analysis Design Framework
- abstract interpretation
- static analysis design = designing a semantics
- so powerful a mindset, a great armor

Static Analysis Engineering Framework
- localizations in space and time are must
- “sparse analysis” framework without accuracy compromise
- can analyze 1MLoC C “in detail”, soundly and globally
- the a.i. mindset helps us a lot throughout our hacking

Static Analysis of Multi-Staged Programs
- MSP is common in mobile/web scripting
- a static typing that respects the practice(Lisp)
- a general static analysis framework via unstaging
- waiting for tests in practice
• Static Analysis Overview
• Static Analysis Design Framework
  • abstract interpretation
  • static analysis design = designing a semantics
  • so powerful a mindset, a great armor
• Static Analysis Engineering Framework
  • localizations in space and time are must
  • “sparse analysis” framework without accuracy compromise
  • can analyze 1MLoC C “in detail”, soundly and globally
  • the a.i. mindset helps us a lot throughout our hacking
• Static Analysis of Multi-Staged Programs
  • MSP is common in mobile/web scripting
  • a static typing that respects the practice(Lisp)
  • a general static analysis framework via unstaging
  • waiting for tests in practice

Thank you