Typing & Static Analysis of Multi-Staged Programs

Kwangkeun Yi

Seoul National University, Korea

5/31/2011 @ UC Berkeley

(co-work with I. Kim, W. Choi, B. Aktemur, C. Calcagno, M. Tatsuda)
We try to help reduce/eliminate errors in software.

- statically: before execution, before sell/embed
- automatically: against explosive sw size
- to find bugs or verify their absence
Our Activities

We published our works in:

- POPL('11, '06), TACAS('11), VMCAI('10, '11), ICSE('11), SAS, ISMM, OOPSLA, FSE, etc.
- TOPLAS, TCS, JFP, SP&E, Acta Informatica, etc.

A commercialization:

Research areas: static analysis, abstract interpretation, programming language theory, type system, theorem proving, model checking, & whatever relevant
Outline

1. Multi-staged Programming
2. Typing Multi-Staged Programs (POPL’06)
3. Static Analysis of Multi-Staged Programs (POPL’11)
Multi-Staged Programming (1/4)

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.
- 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

\[
e ::= \cdots
\]

\[
\begin{align*}
| & \ 'e' \ & \text{code as data} \\
| & \ ,e \ & \text{code substitution} \\
| & \ \text{run } e \ & \text{execute code}
\end{align*}
\]

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

\[
power(x, n) = \text{if } n = 0 \text{ then } 1 \text{ else } x \times power(x, n-1)
\]

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

prepared as

\[
\text{let } spower(n) = \text{if } n = 0 \text{ then } '1' \text{ else } '(x*, (spower (n-1)))'
\]
\[
\text{let } fastpower = '(\lambda x., (spower \ input))'
\]
\[
\text{in } (\text{run fastpower}) \ 2
\]
Practice of Multi-Staged Programming

- open code
  
  '?(x+1)

- intentional variable-capturing substitution
  
  let y = '?(x+1) in '?(λx.,y)

- capture-avoiding substitution
  
  let y = '?(x+1) in '?(λ*x.,y + x)

- imperative operations with open code
  
  cell := '?(x+1); ... 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.
Part I: Our Answer I

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

A Let-Polymorphic Modal Type System for Lisp-style Multi-Staged Programming [Kim, Yi, Calcagno: POPL’06]
Ideas

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

- 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 \( \Box (\Gamma \triangleright int) \)
  - monomorphic store types

Natural ideas worked.
Simple Type System

Type \( A, B \) ::= \( \tau \mid A \to B \mid \Box (\Gamma \triangleright A) \)

code type

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

typing judgment

\( \Gamma_0 \cdots \Gamma_n \vdash e : A \)

(TSBOX)

\[
\begin{align*}
\Gamma_0 \cdots \Gamma_n \Gamma & \vdash e : A \\
\Gamma_0 \cdots \Gamma_n & \vdash \text{box } e : \Box (\Gamma \triangleright A)
\end{align*}
\]

(TSUNBOX)

\[
\begin{align*}
\Gamma_0 \cdots \Gamma_n & \vdash e : \Box (\Gamma_{n+k} \triangleright A) \\
\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} & \vdash \text{unbox}_k e : A
\end{align*}
\]

(TSEVAL)

\[
\begin{align*}
\Gamma_0 \cdots \Gamma_n & \vdash e : \Box (\emptyset \triangleright A) \\
\Gamma_0 \cdots \Gamma_n & \vdash \text{run } e : A
\end{align*}
\]

(for alpha-equiv. at stage 0)
Polymorphic Type System (1/2)

A combination of
- ML’s let-polymorphism
  - syntactic value restriction + multi-staged “expansive$^n(e)$”
  - expansive$^n(e) = False$
  \[\implies e\] never expands the store during its eval. at \(\forall\text{stages} \leq n\)
  
  e.g.) `(\(\lambda x., e\)) : \text{can be expansive}
  
  `(\(\lambda x.\text{run } y\)) : \text{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:
\[ \Box(\{x : \text{int}\} \rho \triangleright \text{int}) \]

- then-branch: \( \Box(\{x : \text{int}\} \rho' \triangleright \text{int}) \)
- else-branch: \( \Box(\rho'' \triangleright \text{int}) \)

let \( x = 'y \) in ‘(,x + w); ‘((,x 1) + z)
\[ x: \forall \alpha \forall \rho. \Box(\{y : \alpha\} \rho \triangleright \alpha) \]

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

- **Unification:**
  - Rémy’s unification for record type $\Gamma$
  - usual unification for new type terms such as $\Box (\Gamma \triangleright A)$ and $A$ 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
Staged programming “practice” has a sound static type system.
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

How to analyze “run e”, the execution of estimated program texts?

[Choi, Aktemur, Yi, Tatsuda: POPL’11] Static Analysis of Multi-Staged Programs via Unstaging Translation
Problem in Static Anaysis of Staged Programs

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

- The set of possible code for \( x \):
  \[
  \{ '0, '(0+2), '(0+2+2), \cdots \}.
  \]
  
  must first be finitely approximated, e.g., by a grammar:

  \[ S \rightarrow 0 | S+2. \]

- 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

### Staged source

\[ e ::= \begin{array}{l}
\lambda x. e \\
\quad e e \\
\quad x \\
\quad \text{'} e \\
\quad , e \\
\quad \text{run } e
\end{array} \]

### Unstaged target

\[ e ::= \begin{array}{l}
\lambda x. e \\
\quad e e \\
\quad x \\
\quad \{\} \\
\quad e\{x=e\} \\
\quad e \cdot x
\end{array} \]
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.:

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

variable capturing into record passing + lookup:

\[ '(\lambda x., ('(x+1)')) \mapsto \lambda \rho_1 \lambda x. ((\lambda \rho_2. (\rho_2 \cdot x) + 1) (\rho_1\{x = x\})) \]
\[ x ::= \text{'}0\text{'}; \]
\[ \text{repeat} \quad x ::= (',x + 2) \quad \text{until } \text{cond}; \]
\[ \text{run } x\]

\[ x ::= \lambda\rho.\text{'}0\text{'}; \]
\[ \text{repeat} \quad x ::= (\lambda h. (\lambda\rho. (h \rho) + 2)) \quad x \quad \text{until } \text{cond}; \]
\[ x \ {}\]
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 $K(e) \xrightarrow{R;A^*} K'(e')$. 

\[ e \xrightarrow{n} e' \]
\[ \Downarrow \quad \Downarrow \]
\[ e \quad e' \Rightarrow \]
\[ e \xrightarrow{R;A^*} e' \]
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 $\overline{e}$ be, respectively, a staged program and its translated unstaged version. If $\llbracket e \rrbracket \sqsubseteq \pi \llbracket e \rrbracket$ and $\alpha \circ \pi \circ \gamma \sqsubseteq \hat{\pi}$ then $\alpha \llbracket e \rrbracket \sqsubseteq \hat{\pi} \llbracket e \rrbracket$. 
Example (1/5): \( [e] \) staged collecting semantics

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

Collecting semantics \([e] = \)

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

\[
\text{run } x \text{ has } \{ 0, 2, 4, 6, \cdots \}
\]
Example (2/5): $[e]$ 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 $[e] =$

- $x, h$ has $\{\langle \lambda \rho_1.0, \emptyset \rangle, \langle \lambda \rho_2.(h \rho_2)+2, \{h \mapsto \langle \lambda \rho_1.0 \rangle \} \rangle, \ldots \}$
- $\rho_1, \rho_2$ has $\{\}$
- $x \ \{\}$ has $\{0, 2, 4, 6, \ldots \}$
Example (3/5): \( \pi \) projection of collecting semantics

Collecting semantics are aligned:

\[
\begin{bmatrix} e \end{bmatrix} \sqsubseteq \pi \begin{bmatrix} e \end{bmatrix}
\]

\[x, h\] has \(\{\langle \lambda \rho_1.0, \emptyset \rangle, \langle \lambda \rho_2.(h \rho_2)+2, \{h \mapsto \langle \lambda \rho_1.0 \rangle \}\}\}, \cdots\) \hspace{0.5cm} \(\pi \mapsto x\) has \(\{0, (0+2), (0+2+2), \cdots\}\) \hspace{0.5cm} \(\rho_1, \rho_2\) has \(\{\}\)

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

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

\[h \rho \mapsto \pi \mapsto \text{"code-filling by } h\text{"}\]
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 cond;
\[
x \ {} \{ \}
\]

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

\[
x \text{ has } \lambda \rho_1.0
\]

\[
x \text{ has } \lambda \rho_2. (h \rho_2)+2 \quad (h \rho_1) \text{ has } V_1 \rightarrow 0 \mid V_1+2
\]

\[
h \text{ has } \lambda \rho_1.0 \quad x \ {} \{ \} \text{ has } V_2 \rightarrow 0 \mid V_1+2
\]

\[
h \text{ has } \lambda \rho_2. (h \rho_2)+2
\]
Example (5/5): \(\hat{\pi}\) projection of analysis

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

\[
\begin{aligned}
  x & \text{ has } S_1 \to \rho_1 \\
  x & \text{ has } S_2 \to \rho_2(S) \\
  S & \to \rho_1 \mid \rho_2(S) \\
  \end{aligned}
\]

run \(x\) has \(V \to 0 \mid V+2\)

- intuition

  "\(\lambda \rho\)" \(\mapsto\) "code indexed as \(\rho\)"

  "\(h \rho\)" \(\mapsto\) "code-filling by \(h\)"

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

  and was \([e] \sqsubseteq \pi[e]\)

Hence, by the projection theorem, correct:

\[
\alpha[e] \sqsubseteq \hat{\pi}[\hat{e}]
\]
Part II: Conclusion

- semantic-preserving unstaging translation
- sound static analysis framework using the translation

\[ e \in D_S \xleftarrow{\gamma} \hat{D}_S \ni \hat{[e]} \]
\[ e \in D_R \xleftarrow{\gamma} \hat{D}_R \ni \hat{[e]} \]

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

- extend to “string-based” (unstructured) multi-staged programming
- realistic static analyses: e.g. static malware detection
- program logic (e.g. separation logic) for multi-staging
- and any topic ~ multi-staging