Course Outline

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)

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.

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

Stage	Computation	Value
0	usual + code + run	usual + code
> 0	code substitution	code

Multi-Staged Programming (3/4)

In examples, we will use Lisp-style staging constructs + only 2 stages

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
  & | & `e & \text{code as data} \\
  & | & ,e & \text{code substitution} \\
  & | & \texttt{run}\ e & \text{execute code}
\end{array}
$$

- code as a value: `` `(1+1) ``
- code composition: `` let y = `(x+1) in '(λx.,y) ``
- code execution: `run '(1+1)`

Multi-Staged Programming (4/4)

Specializer/Partial evaluator

```
power(x,n) = if n=0 then 1 else x * power(x,n-1)
```

v.s. `power(x,3) = x*x*x`

prepared as

```
let spower(n) = if n=0 then `1 else `(x*,(spower (n-1)))
let fastpower = `(λx.,(spower input))
in (run fastpower) 2
```

Practice of Multi-Staged Programming

- open code

```
‘(x+1)
```

- intentional variable-capturing substitution

$$\texttt{let y = ‘(x+1) in ‘(}\lambda\texttt{x.,y)}$$

- capture-avoiding substitution

$$\texttt{let y = ‘(x+1) in ‘(}\lambda^{*}\texttt{x.,y + x)}$$

- imperative operations with open code

$$\texttt{cell := ‘(x+1);}\ \cdots\ \texttt{cell := ‘(y 1);}$$

Challenge I: Special Static Analysis

A static type system that supports the practice.

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

Previous type systems support only part of the practice.

Challenge II: General Static Analysis

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.

Side: Two Camps of Staging Semantics

- variable-capture allowed at stages > 0 (the practice of 30 yrs)

$$\texttt{let y = `(x+1) in `(}\lambda\texttt{x.,y)}$$

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

$$\texttt{(}\lambda\texttt{x.`x) 1}$$

Part I: Special Static Analysis

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

$$\Box(\Gamma \triangleright \mathit{int})$$

- view the type environment Γ as a record type

$$\Gamma = \{x : \mathit{int},\ y : \mathit{int} \rightarrow \mathit{int}, \cdots\}$$

- 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 Γ in code type $\Box(\Gamma \triangleright \mathit{int})$
  - monomorphic store types

Natural ideas worked.

Simple Type System

$$\textit{Type} \quad A, B \quad ::= \quad \iota \mid A \to B \mid \Box(\Gamma \triangleright A)$$

code type

$$\texttt{`(x+1):}\ \Box(\{x : \textit{int}, \cdots\} \triangleright \textit{int})$$

typing judgment

$$\Gamma_0 \cdots \Gamma_n \vdash e : A$$

$$\texttt{(TSBOX)} \quad \frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\ e : \Box(\Gamma \triangleright A)}$$

$$\texttt{(TSUNBOX)} \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\Gamma_{n+k} \triangleright A)}{\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash \texttt{unbox}_k e : A}$$

$$\texttt{(TSEVAL)} \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\emptyset \triangleright A)}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{run}\ \ e : A}$$

(for alpha-equiv. at stage 0)

Polymorphic Type System (1/2)

A combination of

- ML's let-polymorphism
  - syntactic value restriction + multi-staged "$\text{expansive}^n(e)$"
  - $\text{expansive}^n(e) =$ *False*

    $\Longrightarrow$ e never expands the store during its eval. at $\forall$stages$\leq n$

$$\begin{array}{lll} \text{e.g.)} \quad \text{`}(\lambda x.,e) & : & \text{can be expansive} \\ \quad\quad\; \text{`}(\lambda x.\texttt{run}\, y) & : & \text{unexpansive} \end{array}$$

- Rémy's record types [Rémy 1993]
  - type environments as record types with field addition
  - record subtyping + record polymorphism

Polymorphic Type System (2/2)

- `if` e `then ‘(x+1) else ‘1`: $\boxed{\Box(\{x : \mathit{int}\}\rho \rhd \mathit{int})}$
  - then-branch: $\Box(\{x : int\}\rho' \rhd \mathit{int})$
  - else-branch: $\Box(\rho'' \rhd \mathit{int})$
- `let x = ‘y in ‘(,x + w); ‘((,x 1) + z)`
  $\boxed{\texttt{x}: \forall\alpha\forall\rho.\Box(\{y : \alpha\}\rho \rhd \alpha)}$
  - first x: $\Box(\{y : \mathit{int},\ w : \mathit{int}\}\rho' \rhd \mathit{int})$
  - second x: $\Box(\{y : \mathit{int} \to \mathit{int},\ z : \mathit{int}\}\rho'' \rhd \mathit{int} \to \mathit{int})$

Type Inference Algorithm

- Unification:
  - Rémy's unification for record type Γ
  - usual unification for new type terms such as $\Box(\Gamma \triangleright A)$ and $A\,\mathrm{ref}$
- 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

Part II: General Static Analysis (rephrase)

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

Problem in Static Anaysis of Staged Programs

```
x := `0;
repeat x := `(,x + 2)
until cond;
run x
```

- The set of possible code for x:

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

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

$$S \rightarrow \texttt{0} \mid S\texttt{+2}.$$

- 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 $\longmapsto$ Unstaged target

Staged source:

$$
\begin{array}{lcl}
e & ::= & \lambda x.e \\
 & | & e\ e \\
 & | & x \\
 & | & \text{`}\,e \\
 & | & ,e \\
 & | & \texttt{run}\ e
\end{array}
$$

Unstaged target:

$$
\begin{array}{lcl}
e & ::= & \lambda x.e \\
 & | & e\ e \\
 & | & x \\
 & | & \{\} \\
 & | & e\{\texttt{x}=e\} \\
 & | & e \cdot \texttt{x}
\end{array}
$$

Translation Ideas (1/2)

- code into env-taking function:

$$\texttt{`(1+1)} \longmapsto \lambda\rho.\texttt{1+1}$$

- free variable in a code into record lookup:

$$\texttt{`(}x\texttt{+1)} \longmapsto \lambda\rho.(\rho\cdot\texttt{x}) + 1$$

- `run` expression into an application:

$$\texttt{run `(1+1)} \longmapsto (\lambda\rho.\texttt{1+1})\{\}$$

Translation Ideas (2/2)

- code composition into an app. whose actual param. is for the code-to-be-plugged expr.:

$$\texttt{`(,}y\texttt{ + 2)} \longmapsto \texttt{(}\lambda h\texttt{.(}\lambda\rho\texttt{.(}h\ \ \rho\texttt{)+2))}\ y$$

- variable capturing into record passing+lookup:

$$\texttt{'(}\lambda x\texttt{.,(`(}x\texttt{+1)))} \longmapsto \lambda\rho_1\lambda x\texttt{.((}\lambda\rho_2\texttt{.(}\rho_2\cdot\texttt{x)+1)}\ \ \texttt{(}\rho_1\{\texttt{x} = x\}\texttt{))}$$

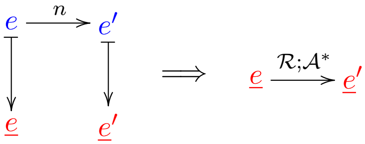
Translation Example

x `:=` `'0;`		x `:=` $\lambda\rho.$`0;`
`repeat`		`repeat`
$\quad x := {}^{\backprime}(,x + 2)$	$\longmapsto$	$\quad x$ `:=` `(`$\lambda h.$`(`$\lambda\rho.$`(`$h\ \rho$`)+2))` x
`until` $cond$`;`		`until` $cond$`;`
`run` x		$x\ \{\}$

Simulation

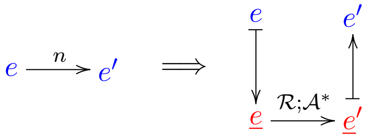
Theorem

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



Theorem

(Inversion) Let e be a $\lambda_{\mathcal{S}}$ expression and R be an environment stack. If $R \vdash e \mapsto (\underline{e}, K)$, then $H \vdash \underline{e} \mapsto e$ for any H such that $\overline{K} \subseteq H$.



Analysis and Projection

$$\begin{array}{ccccc} e & \quad & [\![e]\!] \in D_S & \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} & \hat{D}_S \ni \widehat{[\![e]\!]} \\ \downarrow & & \uparrow \pi & & \uparrow \hat{\pi} \\ \underline{e} & & [\![\underline{e}]\!] \in D_R & \underset{\underline{\alpha}}{\overset{\underline{\gamma}}{\leftrightarrows}} & \hat{D}_R \ni \widehat{[\![\underline{e}]\!]} \end{array}$$

Theorem

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

Example (1/5): $[\![e]\!]$ staged collecting semantics

$$
\begin{array}{l}
x \texttt{ := `0;} \\
\texttt{repeat} \\
\quad x := \text{`}(,x + 2) \\
\texttt{until } cond; \\
\texttt{run } x
\end{array}
$$

Collecting semantics $[\![e]\!] =$

$$
\begin{array}{rll}
x & \text{has} & \{\texttt{`0}, \texttt{`(0+2)}, \texttt{`(0+2+2)}, \cdots\} \\
\texttt{run } x & \text{has} & \{0, 2, 4, 6, \cdots\}
\end{array}
$$

Example (2/5): $[\![\underline{e}]\!]$ unstaged collecting semantics

```
x := λρ1.0;
repeat
    x := (λh.(λρ2.(h ρ2)+2)) x
until cond;
x {}
```

Collecting semantics $[\![\underline{e}]\!] =$

$$
\begin{array}{rll}
x, h & \text{has} & \{\langle \lambda\rho_1.0, \emptyset\rangle, \langle \lambda\rho_2.(h\ \ \rho_2)\texttt{+}2, \{h \mapsto \langle \lambda\rho_1.0\rangle\}\rangle, \cdots\} \\
\rho_1, \rho_2 & \text{has} & \{\} \\
x\ \ \{\} & \text{has} & \{0, 2, 4, 6, \cdots\}
\end{array}
$$

Example (3/5): π projection of collecting semantics

Collecting semantics are aligned:

$$[\![e]\!] \sqsubseteq \pi[\![\underline{e}]\!]$$

$$\begin{array}{lll} x, h & \text{has} & \{\langle \lambda\rho_1.\mathtt{0}, \emptyset\rangle, \\ & & \;\langle \lambda\rho_2.\mathtt{(}h\ \ \rho_2\mathtt{)+2}, \\ & & \;\;\{h \mapsto \langle \lambda\rho_1.\mathtt{0}\rangle\}\rangle, \\ & & \cdots\} \\ \rho_1, \rho_2 & \text{has} & \{\} \end{array} \quad \stackrel{\pi}{\longmapsto} \quad x \quad \text{has} \quad \begin{array}{l} \{\text{`}\mathtt{0}, \text{`}\mathtt{(0+2)}, \\ \;\;\text{`}\mathtt{(0+2+2)}, \cdots\} \end{array}$$

- π = inverse translation + removing admin stuff
- intuition

$$\begin{array}{lcl} \text{“}\lambda\rho\text{”} & \stackrel{\pi}{\longmapsto} & \text{“code indexed as } \rho\text{”} \\ \text{“}h\ \rho\text{”} & \stackrel{\pi}{\longmapsto} & \text{“code-filling by } h\text{”} \end{array}$$

Example (4/5): $[\![\hat{\underline{e}}]\!]$ unstaged conventional analysis

$$
\begin{array}{l}
x \;\texttt{:=}\; \lambda\rho_1\texttt{.0;} \\
\texttt{repeat} \\
\quad x \;\texttt{:=}\; (\lambda h.(\lambda\rho_2.(h\ \rho_2)\texttt{+2}))\ x \\
\texttt{until}\ cond; \\
x\ \{\}
\end{array}
$$

0-CFA analysis $[\![\hat{\underline{e}}]\!]$ in set-constraint style

$$
\begin{array}{lll}
x & \text{has} & \lambda\rho_1.0 \\
x & \text{has} & \lambda\rho_2.(h\ \rho_2)+2 \\
h & \text{has} & \lambda\rho_1.0 \\
h & \text{has} & \lambda\rho_2.(h\ \rho_2)+2
\end{array}
\qquad
\begin{array}{rll}
(h\ \rho_2) & \text{has} & V_1 \to 0 \mid V_1+2 \\
x\ \{\} & \text{has} & V_2 \to 0 \mid V_1+2
\end{array}
$$

Example (5/5): $\hat{\pi}$ projection of analysis

$$\begin{array}{rll} x & \text{has} & \lambda\rho_1.0 \\ x & \text{has} & \lambda\rho_2.(h\ \rho_2)+2 \\ h & \text{has} & \lambda\rho_1.0 \\ h & \text{has} & \lambda\rho_2.(h\ \rho_2)+2 \\ x\ \{\} & \text{has} & V \to 0 \mid V+2 \end{array} \quad \overset{\hat{\pi}}{\longmapsto} \quad \begin{array}{rll} x & \text{has} & S_1 \to \rho_1 \\ x & \text{has} & S_2 \to \rho_2(S) \\ & & S \to \rho_1 \mid \rho_2(S) \\ \texttt{run}\ x & \text{has} & V \to 0 \mid V+2 \end{array}$$

- intuition

$$\begin{array}{rcl} \text{“}\lambda\rho\text{”} & \overset{\hat{\pi}}{\longmapsto} & \text{“code indexed as } \rho\text{”} \\ \text{“}h\ \rho\text{”} & \overset{\hat{\pi}}{\longmapsto} & \text{“code-filling by } h\text{”} \end{array}$$

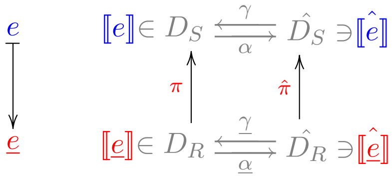
- $\hat{\pi}$ satisfies the safety condition: $\alpha \circ \pi \circ \underline{\gamma} \sqsubseteq \hat{\pi}$
- and was $[\![e]\!] \sqsubseteq \pi[\![\underline{e}]\!]$

Hence, by the projection theoreom, correct:

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

Summary: General Static Analysis Framework for MSP

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



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

Messages

- 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

Messages

- 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