Typing & Static Analysis of Multi-Staged Programs

Kwangkeun Yi

School of Computer Science & Engineering
Seoul National University

1/10/2012 @ NII Tokyo

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

ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

About Us

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

ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

Our Activities

We published our works in:

- POPL('11, '06), TACAS('11), VMCAI('12, '11, '10), ICSE('11), SAS, ISMM, OOPSLA, FSE, etc.
- TOPLAS, TCS, JFP, SP&E, Acta Informatica, etc.

A commercialization:

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

ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

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.

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

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

- 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

$$\texttt{`(x+1)}$$

- 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

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

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

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

- view the type environment Γ as a record type

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

- 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

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

typing judgment

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

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

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

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

(for alpha-equiv. at stage 0)

Polymorphic Type System (1/2)

A combination of

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

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

e.g.)	${}^{\backprime}(\lambda x.,e)$	:	can be expansive
	${}^{\backprime}(\lambda x.\mathtt{run}\,y)$	:	unexpansive

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

ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

Polymorphic Type System (2/2)

- `if` e `then ‘(x+1) else ‘1`: $\boxed{\Box(\{x : int\}\rho \triangleright int)}$
  - then-branch: $\Box(\{x : int\}\rho' \triangleright int)$
  - else-branch: $\Box(\rho'' \triangleright int)$

- `let x = ‘y in ‘(,x + w); ‘((,x 1) + z)`

  x: $\boxed{\forall\alpha\forall\rho.\Box(\{y : \alpha\}\rho \triangleright \alpha)}$
  - first x: $\Box(\{y : int,\ w : int\}\rho' \triangleright int)$
  - second x: $\Box(\{y : int \to int,\ z : int\}\rho'' \triangleright int \to int)$

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 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 I: Message

Staged programming "practice" has a sound static type system.

Part II: Challenge (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: POPL'11] Static Analysis of Multi-Staged Programs via Unstaging Translation

ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

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:

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

- 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

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

$\longmapsto$

Unstaged target

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

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.:

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

- variable capturing into record passing+lookup:

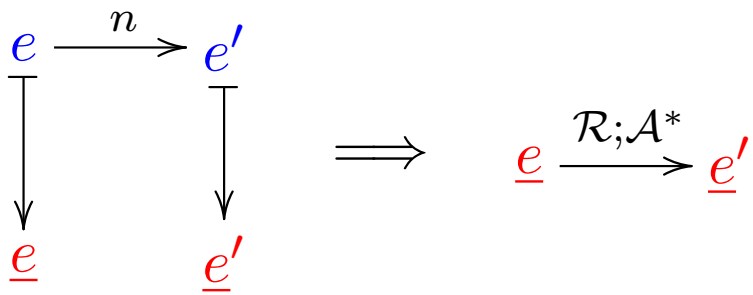
$$\texttt{'(}\lambda x.\texttt{,(`(}x\texttt{+1)))} \longmapsto \lambda \rho_1 \lambda x.((\lambda \rho_2.(\rho_2 \cdot \texttt{x})\texttt{+1})\ (\rho_1\{\texttt{x} = x\}))$$

Translation Example

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

Theorem

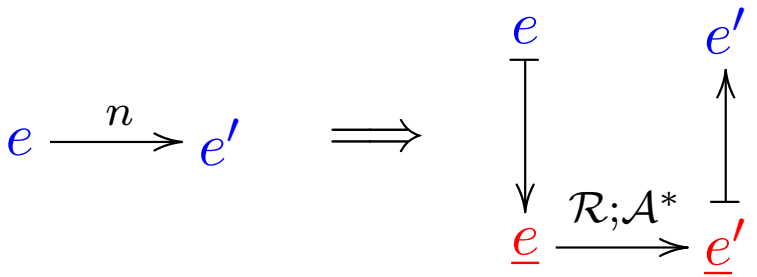
(Simulation) Let e be a stage-n λ_S expression with no free variables such that $e \xrightarrow{n} e'$. Let $R \vdash e \mapsto (\underline{e}, K)$ and $R \vdash e' \mapsto (\underline{e}', K')$. Then $K(\underline{e}) \xrightarrow{\mathcal{R};\mathcal{A}^*} K'(\underline{e}')$.



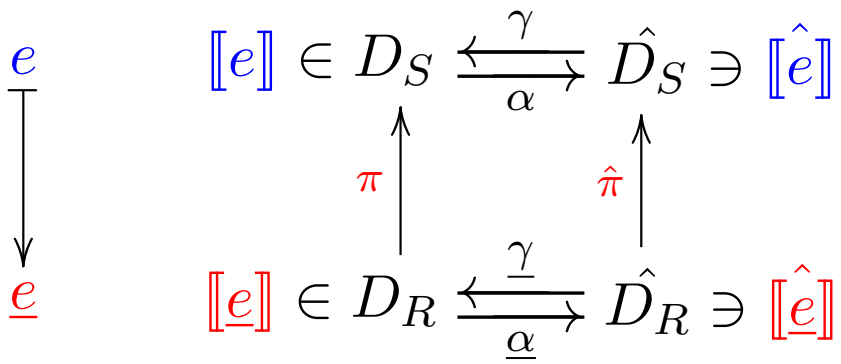
Inversion

Theorem

(Inversion) Let e be a λ_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



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

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

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

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

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

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

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

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

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

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

Collecting semantics are aligned:

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

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

- π = inverse translation + removing admin stuff
- intuition

$$\text{“}\lambda\rho\text{”} \overset{\pi}{\longmapsto} \text{“code indexed as } \rho\text{”}$$
$$\text{“}h\ \rho\text{”} \overset{\pi}{\longmapsto} \text{“code-filling by } h\text{”}$$

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

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

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

x has $\lambda\rho_1.0$
x has $\lambda\rho_2.(h\ \rho_2)\texttt{+}2$
h has $\lambda\rho_1.0$
h has $\lambda\rho_2.(h\ \rho_2)\texttt{+}2$

$(h\ \rho_2)$ has $V_1 \to 0 \mid V_1\texttt{+}2$
$x\ \{\}$ has $V_2 \to 0 \mid V_1\texttt{+}2$

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

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

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

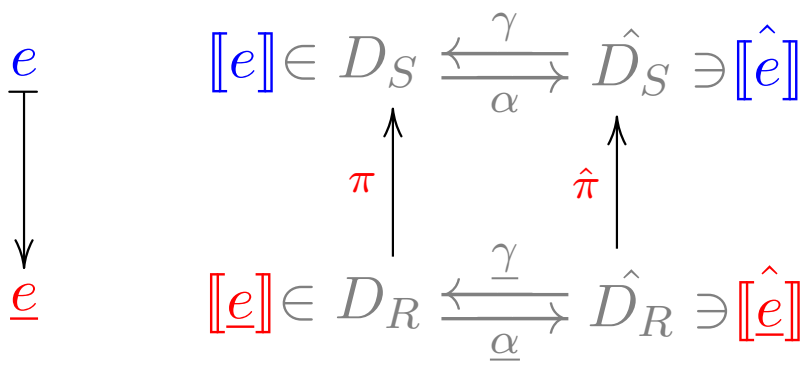
Hence, by the projection theoreom, correct:

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

ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

Part II: Conclusion

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



unstaging + usual static analysis + projection are sufficient.

Questions and Things to Do

- why not directly on staged programs?
  - typing worked because?
  - how for more than typing?
- extend to "string-based" (unstructured) multi-staged programming
- program logic (e.g. separation logic) for multi-staging
- realistic static analyses
  - e.g. static Javascript malware detection