

# Typing & Static Analysis of Multi-Staged Programs

Kwangkeun Yi

School of Computer Science & Engineering  
Seoul National University

6/7/2011 @ Oxford U

(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*

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

$$e ::= \dots$$

	<code>'e</code>	code as data
	<code>,e</code>	code substitution
	<code>run e</code>	execute code

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



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 = '(\x.,(spower input))
in (run fastpower) 2
```





# Practice of Multi-Staged Programming

- open code

`'(x+1)`

- intentional variable-capturing substitution

`let y = '(x+1) in '(\x.,y)`

- capture-avoiding substitution

`let y = '(x+1) in '(\*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.



# 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.

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]

# Comparison

- |                           |                             |
|---------------------------|-----------------------------|
| (1) closed code and eval  | (2) open code               |
| (3) imperative operations | (4) type inference          |
| (5) var-capturing subst.  | (6) capture-avoiding subst. |
| (7) polymorphism          |                             |

## Our system

[Rhiger 2005]	+1	+2	+3	-4	+5	-6	-7
[Calcagno et al. 2004]	+1	+2	-3	+4	-5	+6	+7
[Ancona & Moggi 2004]	+1	+2	+3	-4	-5	+6	-7
[Taha & Nielson 2003]	+1	+2	-3	-4	-5	+6	+7
[Chen & Xi 2003]	+1	+2	+3	-4	+5	-6	+7
[Nanevsky & Pfenning 2002]	+1	+2	+3	-4	-5	+6	-7
MetaML/Ocaml[2000,2001]	+1	+2	-3	+4	-5	+6	+7
[Davies 1996]	-1	+2	-3	-4	-5	+6	-7
[Davies & Pfenning 1996,2001]	+1	-2	+3	+4	-5	+6	-7

- code's type: parameterized by its expected context

$$\Box(\Gamma \triangleright int)$$

- view the type environment  $\Gamma$  as a record type

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

- 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  $A, B ::= \iota \mid A \rightarrow B \mid \Box(\Gamma \triangleright A)$

code type

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

typing judgment

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

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

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

(TSEVAL) 
$$\frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\emptyset \triangleright A)}{\Gamma_0 \cdots \Gamma_n \vdash \text{run } e : A}$$
 (for alpha-equiv. at stage 0)



A combination of

- ML's let-polymorphism
  - syntactic value restriction + multi-staged “ $\text{expansive}^n(e)$ ”
  - $\text{expansive}^n(e) = \text{False}$   
 $\implies e$  never expands the store during its eval. at  $\forall \text{stages} \leq n$

e.g.)  $(\lambda x. e)$  : can be expansive  
 $(\lambda x. \text{run } y)$  : unexpansive

- 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  $'1$ :  $\boxed{\square(\{x : int\}\rho \triangleright int)}$ 
  - then-branch:  $\square(\{x : int\}\rho' \triangleright int)$
  - else-branch:  $\square(\rho'' \triangleright int)$
- let  $x = 'y$  in  $'(, x + w)$ ;  $'((, x 1) + z)$   
 $\boxed{x: \forall \alpha \forall \rho. \square(\{y : \alpha\}\rho \triangleright \alpha)}$ 
  - first  $x$ :  $\square(\{y : int, w : int\}\rho' \triangleright int)$
  - second  $x$ :  $\square(\{y : int \rightarrow int, z : int\}\rho'' \triangleright int \rightarrow int)$

- Unification:
  - Rémy's unification for record type  $\Gamma$
  - usual unification for new type terms such as  $\square(\Gamma \triangleright A)$  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

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

# Problem in Static Analysis of Staged Programs

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

- The set of possible code for  $x$ :

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

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

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

- analyzing “`run x`” needs code, not the grammar.



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.

Staged source

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

$\mapsto$

Unstaged target

$$e ::= \lambda x.e$$
$$| e e$$
$$| x$$
$$| \{$$
$$| e\{x=e\}$$
$$| e \cdot x$$

# Translation Ideas (1/2)

- code into env-taking function:

$$\text{'(1+1)} \mapsto \lambda\rho.1+1$$

- free variable in a code into record lookup:

$$\text{'(x+1)} \mapsto \lambda\rho.(\rho \cdot x) + 1$$

- run expression into an application:

$$\text{run ' (1+1)} \mapsto (\lambda\rho.1+1)\{\}$$



# Translation Ideas (2/2)

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

$$\text{'(, } y + 2) \mapsto (\lambda h. (\lambda \rho. (h \ \rho) + 2)) \ y$$

- variable capturing into record passing+lookup:

$$\text{'}(\lambda x. , (\text{'( } x + 1) )) \mapsto \lambda \rho_1 \lambda x. ((\lambda \rho_2. (\rho_2 \cdot x) + 1) \ (\rho_1 \{x = x\}))$$



# Translation Example

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



## Theorem

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

$$\begin{array}{ccc} e & \xrightarrow{n} & e' \\ \downarrow & & \downarrow \\ \underline{e} & & \underline{e}' \end{array} \quad \Longrightarrow \quad \underline{e} \xrightarrow{\mathcal{R}; \mathcal{A}^*} \underline{e}'$$

## Theorem

(Inversion) Let  $e$  be a  $\lambda_S$  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  $\overline{K} \subseteq H$ .

$$e \xrightarrow{n} e' \quad \Longrightarrow \quad \begin{array}{ccc} e & & e' \\ \downarrow & & \uparrow \\ \underline{e} & \xrightarrow{\mathcal{R}; \mathcal{A}^*} & \underline{e}' \end{array}$$

# Analysis and Projection

$$\begin{array}{ccc} e & & [e] \in D_S \xleftrightarrow[\alpha]{\gamma} \hat{D}_S \ni [\hat{e}] \\ \downarrow & & \uparrow \pi \qquad \qquad \uparrow \hat{\pi} \\ \underline{e} & & [\underline{e}] \in D_R \xleftrightarrow[\underline{\alpha}]{\underline{\gamma}} \hat{D}_R \ni [\underline{\hat{e}}] \end{array}$$

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

# Example (1/5): $\llbracket e \rrbracket$ staged collecting semantics

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

Collecting semantics  $\llbracket e \rrbracket =$

```
 $x$  has  $\{ '0, '(0+2), '(0+2+2), \dots \}$   
run  $x$  has  $\{ 0, 2, 4, 6, \dots \}$ 
```

## Example (2/5): $\llbracket e \rrbracket$ unstaged collecting semantics

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

Collecting semantics  $\llbracket e \rrbracket =$

$x, h$  has  $\{\langle \lambda\rho_1.0, \emptyset \rangle, \langle \lambda\rho_2. (h \ \rho_2)+2, \{h \mapsto \langle \lambda\rho_1.0 \rangle\} \rangle, \dots\}$   
 $\rho_1, \rho_2$  has  $\{\}$   
 $x \ \{\}$  has  $\{0, 2, 4, 6, \dots\}$

# Example (3/5): $\pi$ projection of collecting semantics

Collecting semantics are aligned:

$$\llbracket e \rrbracket \sqsubseteq \pi \llbracket e \rrbracket$$

$$\begin{array}{l} x, h \text{ has } \{ \langle \lambda \rho_1.0, \emptyset \rangle, \\ \quad \langle \lambda \rho_2.(h \ \rho_2)+2, \\ \quad \{ h \mapsto \langle \lambda \rho_1.0 \rangle \} \rangle, \\ \quad \dots \} \\ \rho_1, \rho_2 \text{ has } \{ \} \end{array} \quad \xrightarrow{\pi} \quad x \text{ has } \{ '0, '(0+2), \\ \quad '(0+2+2), \dots \}$$

- $\pi$  = inverse translation + removing admin stuff
- intuition

$$\begin{array}{l} \text{"}\lambda\rho\text{"} \xrightarrow{\pi} \text{"code indexed as } \rho\text{"} \\ \text{"}h \ \rho\text{"} \xrightarrow{\pi} \text{"code-filling by } h\text{"} \end{array}$$



# Example (4/5): $\llbracket \hat{e} \rrbracket$ 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  $\llbracket \hat{e} \rrbracket$  in set-constraint style

$x$	has	$\lambda \rho_1.0$		
$x$	has	$\lambda \rho_2. (h \rho_2)+2$	$(h \rho_1)$	has $V_1 \rightarrow 0 \mid V_1+2$
$h$	has	$\lambda \rho_1.0$	$x \{ \}$	has $V_2 \rightarrow 0 \mid V_1+2$
$h$	has	$\lambda \rho_2. (h \rho_2)+2$		

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

$x$	has	$\lambda\rho_1.0$		$x$	has	$S_1 \rightarrow \rho_1$
$x$	has	$\lambda\rho_2.(h \rho_2)+2$	$\xrightarrow{\hat{\pi}}$	$x$	has	$S_2 \rightarrow \rho_2(S)$
$h$	has	$\lambda\rho_1.0$				$S \rightarrow \rho_1 \mid \rho_2(S)$
$h$	has	$\lambda\rho_2.(h \rho_2)+2$		run $x$	has	$V \rightarrow 0 \mid V+2$
$x \{ \}$	has	$V \rightarrow 0 \mid V+2$				

- intuition

“ $\lambda\rho$ ”  $\xrightarrow{\hat{\pi}}$  “code indexed as  $\rho$ ”

“ $h \rho$ ”  $\xrightarrow{\hat{\pi}}$  “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:

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

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

$$\begin{array}{ccc} e & \llbracket e \rrbracket \in D_S & \xleftrightarrow[\alpha]{\gamma} \hat{D}_S \ni \llbracket \hat{e} \rrbracket \\ \downarrow & \uparrow \pi & \uparrow \hat{\pi} \\ \underline{e} & \llbracket \underline{e} \rrbracket \in D_R & \xleftrightarrow[\alpha]{\gamma} \hat{D}_R \ni \llbracket \hat{\underline{e}} \rrbracket \end{array}$$

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

