Typing Multi-Staged Programs and Beyond

Kwangkeun Yi

Research on Software Analysis for Error-free Computing Center
Seoul National University

12/3/2010 @ Tsinghua Univ., Beijing

(co-work with Iksoon Kim, Cristiano Calcagno, Wontai Choe, Baris Aktemur, Makoto Tatsuda)

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

About Us

We want 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

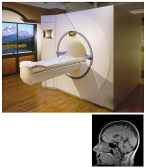
Our approach:

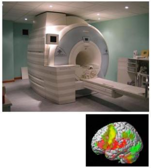
- "semantics-based static analysis"
- and lots of engineering

Our Goals

R&D of static software analysis tools:

"SW MRI" "SW fMRI" "SW PET"





Our Activities

We publicize our works in:

- POPL('06, '11), CAV('08), VMCAI('10, '11), ICSE('11), SAS, OOPSLA, FSE, etc.
- ACM TOPLAS, TCS, JFP, SP&E, Acta Informatica, etc.

A commercialization:

Sparrow
The Early Bird

Research areas: *static analysis, abstract interpreation, programming languge theory, type system, theorem proving, model checking, & whatever relevant* (don't care: orthodox or unorthodox)

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

Outline

Multi-Staged Programming (1/2)

program texts (code) as first class objects
"meta programming"

A general concept that subsumes

- macros
- Lisp/Scheme's quasi-quotation
- partial evaluation
- runtime code generation

Common in mainstream pgm'ng: Lisp/Scheme, C's macros, C++'s templates, C#, JavaScript, PHP, Python, etc.

Multi-Staged Programming (2/2)

- 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 + eval	usual + code
> 0	code substitution	code

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

Multi-Staged Programming Examples (1/2)

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

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

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

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

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

Code as data

```
let NULL = `0
let body = `(if e = ,NULL then abort() ...)
in eval body
```

Multi-Staged Programming Examples (2/2)

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 fastpower10 = eval `(λx.,(spower 10))
in fastpower10 2
```

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

Review: Practice of Multi-Staged Programming

Features of Lisp/Scheme's quasi-quotation system

Review: Practice of Multi-Staged Programming

- open code

```
‘(x+1)
```

Features of Lisp/Scheme's quasi-quotation system

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

Review: Practice of Multi-Staged Programming

- open code

```
`(x+1)
```

- intentional variable-capturing substitution at stages > 0

$$`(\lambda x.,(\texttt{spower 10}))$$

Features of Lisp/Scheme's quasi-quotation system

- open code

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

- intentional variable-capturing substitution at stages > 0

$$\texttt{`(}\lambda\texttt{x.,(spower 10))}$$

- capture-avoiding substitution

$$\texttt{`(}\lambda^{*}\texttt{x.,(spower 10) + x)}$$

Features of Lisp/Scheme's quasi-quotation system

Review: Practice of Multi-Staged Programming

- open code

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

- intentional variable-capturing substitution at stages > 0

$$\texttt{`(}\lambda\texttt{x.,(spower 10))}$$

- capture-avoiding substitution

$$\texttt{`(}\lambda^{*}\texttt{x.,(spower 10) + x)}$$

- imperative operations with open code

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

Features of Lisp/Scheme's quasi-quotation system

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

Challenge I

A static type system that supports the practice.

Should allow programmers both

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

Existing 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
- breaking the basic assumption of conventional static analysis

No general static analysis method yet.

Our Response I

A type system for ML + Lisp's quasi-quote system

- supports 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

Our Response II

A static analysis framework = unstaging-translate; analyze; project

apply an unstaging translation;

apply conventional static analysis techniques;

cast the analysis result back in terms of multi-staged programs

Theory

- the unstaing translation is “correct”
- a safe condition for the projection operation

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

A Let-Polymorphic Modal Type System for Lisp-style MSP [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	$+1$	$+2$	$+3$	$+4$	$+5$	$+6$	$+7$
[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

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

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.

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

Multi-Staged Language

$$
\begin{array}{llll}
e ::= & c \mid x \mid \lambda x.e \mid e\ e & & \\
\mid & \texttt{box}\ e & \text{code as data} & `\ e \\
\mid & \texttt{unbox}_k\ e & \text{code substitution} & , \ldots , e \\
\mid & \texttt{eval}\ e & \text{execute code} & \\
\mid & \lambda^* x.e & \text{gensym} & \\
\mid & \ldots & &
\end{array}
$$

Evaluation

$$\mathcal{E} \vdash e \xrightarrow{n} r$$

where

$\mathcal{E}$: value environment

n: a stage number

r: a value or `err`

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

Operational Semantics (stage $n \geq 0$)

- at stage 0: normal evaluation + code + eval
- at stage > 0: code substitution

$$(\mathtt{EBOX}) \qquad \frac{\mathcal{E} \vdash e \xrightarrow{n+1} v}{\mathcal{E} \vdash \mathtt{box}\ e \xrightarrow{n} \mathtt{box}\ v}$$

$$(\mathtt{EUNBOX}) \qquad \frac{\mathcal{E} \vdash e \xrightarrow{0} \mathtt{box}\ v \qquad k > 0}{\mathcal{E} \vdash \mathtt{unbox}_k\ e \xrightarrow{k} v}$$

$$(\mathtt{EEVAL}) \qquad \frac{\mathcal{E} \vdash e \xrightarrow{0} \mathtt{box}\ v \qquad \mathcal{E} \vdash v \xrightarrow{0} v'}{\mathcal{E} \vdash \mathtt{eval}\ e \xrightarrow{0} v'}$$

Simple Type System (1/2)

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

code type

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

typing judgment

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

Simple Type System (1/2)

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

code type

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

typing judgment

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

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

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

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

(for alpha-equiv. at stage 0)

Simple Type System (2/2)

(TSCON) $\Gamma_0 \cdots \Gamma_n \vdash c : \iota$

(TSVAR) $$\frac{\Gamma_n(x) = A}{\Gamma_0 \cdots \Gamma_n \vdash x : A}$$

(TSABS) $$\frac{\Gamma_0 \cdots (\Gamma_n + x : A) \vdash e : B}{\Gamma_0 \cdots \Gamma_n \vdash \lambda x.e : A \to B}$$

(TSGENSYM) $$\frac{\Gamma_0 \cdots (\Gamma_n + w : A) \vdash [x^n \overset{n}{\mapsto} w]\, e : B \quad \text{fresh } w}{\Gamma_0 \cdots \Gamma_n \vdash \lambda^* x.e : A \to B}$$

(TSAPP) $$\frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A \to B \qquad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A}{\Gamma_0 \cdots \Gamma_n \vdash e_1 e_2 : B}$$

Polymorphic Type System (1/4)

A combination of

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

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

$$\begin{array}{lllll} \text{e.g.)} & \text{`}(\lambda x.\text{,}e) & : & \text{can be expansive} \\ & \text{`}(\lambda x.\mathtt{eval}\ y) & : & \text{unexpansive} \end{array}$$

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

Polymorphic Type System (2/4)

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

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

Polymorphic Type System (3/4)

typing judgment

$$\Delta_0 \cdots \Delta_n \vdash e : A$$

(TBOX)
$$\frac{\Delta_0 \cdots \Delta_n \Gamma \vdash e : A}{\Delta_0 \cdots \Delta_n \vdash \texttt{box}\, e : \Box(\Gamma \triangleright A)}$$

(TUNBOX)
$$\frac{\Delta_0 \cdots \Delta_n \vdash e : \Box(\Gamma \triangleright A) \quad \Delta_{n+k} \succ \Gamma \quad k > 0}{\Delta_0 \cdots \Delta_n \cdots \Delta_{n+k} \vdash \texttt{unbox}_k\ e : A}$$

(TEVAL)
$$\frac{\Delta_0 \cdots \Delta_n \vdash e : \Box(\emptyset \triangleright A)}{\Delta_0 \cdots \Delta_n \vdash \texttt{eval}\ \ e : A}$$

Polymorphic Type System (4/4)

(TVAR) $$\frac{\Delta_n(x) \succ A}{\Delta_0 \cdots \Delta_n \vdash x : A}$$

(TABS) $$\frac{\Delta_0 \cdots (\Delta_n + x : A) \vdash e : B}{\Delta_0 \cdots \Delta_n \vdash \lambda x.e : A \to B}$$

(TAPP) $$\frac{\Delta_0 \cdots \Delta_n \vdash e_1 : A \to B \qquad \Delta_0 \cdots \Delta_n \vdash e_2 : A}{\Delta_0 \cdots \Delta_n \vdash e_1 e_2 : B}$$

(TLETIMP) $$\frac{\begin{array}{l}\text{expansive}^n(e_1) \\ \Delta_0 \cdots \Delta_n \vdash e_1 : A \qquad \Delta_0 \cdots \Delta_n + x : A \vdash e_2 : B\end{array}}{\Delta_0 \cdots \Delta_n \vdash \texttt{let}\ (x\ e_1)\ e_2 : B}$$

(TLETAPP) $$\frac{\begin{array}{l}\neg\,\text{expansive}^n(e_1) \\ \Delta_0 \cdots \Delta_n \vdash e_1 : A \\ \Delta_0 \cdots \Delta_n + x : \mathit{GEN}_A(\Delta_0 \cdots \Delta_n) \vdash e_2 : B\end{array}}{\Delta_0 \cdots \Delta_n \vdash \texttt{let}\ (x\ e_1)\ e_2 : B}$$

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

Type Inference Algorithm

- Unification:
  - Rémy's unification for record type Γ
  - usual unification for new type terms such as $\Box(\Gamma \triangleright A)$ and $A\,\mathrm{ref}$
- Type inference algorithm:
  - the same structure as top-down version $\mathcal{M}$ [Lee and Yi 1998] of the $\mathcal{W}$
  - usual on-the-fly instantiation and unification

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
- Type inference algorithm:
  - the same structure as top-down version $\mathcal{M}$ [Lee and Yi 1998] of the $\mathcal{W}$
  - usual on-the-fly instantiation and unification

Sound If $\texttt{infer}(\emptyset, e, \alpha) = S$ then $\emptyset; \emptyset \vdash e : S\alpha$.

Complete If $\emptyset; \emptyset \vdash e : R\alpha$ then $\texttt{infer}(\emptyset, e, \alpha) = S$ and $R = TS$ for some T.

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

Part I: Conclusion

A type system for multi-staged programming practice (ML + Lisp's quasi-quote)

- conservative extension to ML's let-polymorphism
- principal type inference algorithm

Exact details, lemmas, proof sketchs, and embedding relations in the POPL'06 paper; full proofs in its companion technical report.

Part I: Conclusion

A type system for multi-staged programming practice (ML + Lisp's quasi-quote)

- conservative extension to ML's let-polymorphism
- principal type inference algorithm

Exact details, lemmas, proof sketchs, and embedding relations in the POPL'06 paper; full proofs in its companion technical report.

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

Part II

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

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
- breaking the basic assumption of conventional static analysis

How to statically analyze the semantics of code generated-and-run by the program?

Problem

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

- The set of possible code for x:

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

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

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

- analyzing "run x" requires every code implied by the grammar must be exposed first!?

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

Our Solution

a three-step approach: 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.

Unstaging Translation

The previous example is translated as

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

- Code into env-taking function:

$$\texttt{`0} \longmapsto \lambda\rho.\texttt{0}$$

- The `run` expression into an application:
  $\texttt{run}\;\texttt{`0} \longmapsto (\lambda\rho.\texttt{0})\{\}$
- Free variables in a code into record accesses. $\texttt{`}x \longmapsto \lambda\rho.\rho{\cdot}x$
- Code composition $\texttt{`(,}x\;\texttt{+}\;\texttt{2)}$ into a ftn-generating app. whose actual param. is the part for the code-to-be-plugged expr.:

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

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

$$\underbrace{\texttt{box}\ (\ldots\texttt{unbox}\ (\boxed{\cdot})\ldots\texttt{unbox}\ (\boxed{\cdot})\ldots)}$$

$$(\lambda h_1.(\lambda h_2.(\lambda \rho.\quad \ldots \quad h_1\ \rho \quad \ldots \quad h_2\ \rho \quad \ldots))(\boxed{\cdot}))(\boxed{\cdot})$$

Illustration of the translation of a `box` expression with two `unbox`es.

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 $K(\underline{e}) \xrightarrow{\mathcal{R};\mathcal{A}^*} K'(\underline{e}')$.

$$
\begin{tikzcd}
e \arrow[r, "n"] \arrow[d, mapsto] & e' \arrow[d, mapsto] \\
\underline{e} & \underline{e}'
\end{tikzcd}
\quad \Longrightarrow \quad
\underline{e} \xrightarrow{\mathcal{R};\mathcal{A}^*} \underline{e}'
$$

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 $\overline{K} \subseteq H$.

$$
e \xrightarrow{\;n\;} e' \quad \Longrightarrow \quad
\begin{tikzcd}
e \arrow[d, mapsto] & e' \\
\underline{e} \arrow[r, "\mathcal{R};\mathcal{A}^*"] & \underline{e'} \arrow[u, mapsto]
\end{tikzcd}
$$

Analysis and Projection

$$
\begin{tikzcd}
e \arrow[d, mapsto] & \llbracket e \rrbracket \in D_S \arrow[r, shift right, "\alpha"'] & \hat{D}_S \ni \widehat{\llbracket e \rrbracket} \arrow[l, shift right, "\gamma"'] \\
\underline{e} & \llbracket \underline{e} \rrbracket \in D_R \arrow[u, "\pi"] \arrow[r, shift right, "\underline{\alpha}"'] & \hat{D}_R \ni \widehat{\llbracket \underline{e} \rrbracket} \arrow[u, "\hat{\pi}"'] \arrow[l, shift right, "\underline{\gamma}"']
\end{tikzcd}
$$

Theorem

(Safe Projection) Let e *and* $\underline{e}$ *be, respectively, a staged program and its translated unstaged version. If* $\llbracket e \rrbracket \sqsubseteq \pi \llbracket \underline{e} \rrbracket$ *and* $\alpha \circ \pi \circ \underline{\gamma} \sqsubseteq \hat{\pi}$ *then* $\alpha \llbracket e \rrbracket \sqsubseteq \hat{\pi} \widehat{\llbracket \underline{e} \rrbracket}$.

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

Example (1/2)

After translation:

$$
\begin{array}{l}
x \ \texttt{:=} \ \lambda\rho_1\texttt{.0;} \\
\texttt{repeat} \\
\quad x \ \texttt{:=} \ \texttt{(}\lambda h\texttt{.(}\lambda\rho_2\texttt{.}\underline{\texttt{(}h \ \ \rho_2\texttt{)}}_1 \ \texttt{+ 2))} \ x \\
\texttt{until} \ \ cond\texttt{;} \\
\underline{\texttt{(}x \ \ \{\}\texttt{)}}_2
\end{array}
$$

Analysis: collecting/resolving constraints

$$
\begin{array}{rcl}
V_x & \ni & \lambda\rho_1 \\
V_x & \ni & \lambda\rho_2 \\
V_h & \ni & V_x \\
V_h & \ni & \lambda\rho_1 \\
V_h & \ni & \lambda\rho_2 \\
V_1 & \ni & 0 \\
V_1 & \ni & V_1{+}2 \\
V_2 & \ni & 0 \\
V_2 & \ni & V_1{+}2
\end{array}
$$

then the analysis may conclude

$$
\begin{array}{l}
V_1 \rightarrow 0 \mid V_1{+}2 \\
V_2 \rightarrow 0 \mid V_1{+}2
\end{array}
$$

Example (2/2)

Projection: cast the analysis results back in terms of the original staged program

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

- V_h's values $\lambda\rho_1$ and $\lambda\rho_2$ are projected to code exprs. `` `0 `` and `` `(,x + 2) ``.
- i.e., code to be plugged into the place of "$,x$" can be `` `0 `` and, recursively, `` `(,x + 2) ``.
- Underlying projections satisfy the safety conditions.

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

A static analysis method for multi-staged programs

- semantic-preserving unstaging translation
- sound projection of conventional analysis for unstaged program back in terms of original, staged program

Exact details, lemmas, proof sketchs in the POPL'11 paper; full proofs in its companion technical report.

Part II: Conclusion

A static analysis method for multi-staged programs

- semantic-preserving unstaging translation
- sound projection of conventional analysis for unstaged program back in terms of original, staged program

Exact details, lemmas, proof sketchs in the POPL'11 paper; full proofs in its companion technical report.

Thank you.

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