System Zoo
(work-in-progress)

Kwangkeun Yi

Research On Program Analysis System
National Creative Research Initiative Center
Dept. of Computer Science
KAIST

11/11/2002@SNU
System Zoo

a software tool to make softwares safe
A Shame

unsafe softwares
Unsafe Softwares

- **bugs**: everywhere

- cost: big
  
  - recall $k \times$ million cars/zipels/phones?
  
  - Ariane rocket: 500 million dollars, 2 billion dollars

- mass anxiety $\Rightarrow$ new legislations $\Rightarrow$ insurances $\Rightarrow$ high cost
Technology for Safe Softwares

very primitive the-status-quo

- ad-hoc/cowboy approaches:
  - testing, debugging, code review, simulations, testing, field manual, etc.

- performance:
  - AT&T: productivity = 10 lines/month (1995)
Badly Need Better Technology

difficult/impossible for manual debugging

- complicated\(\infty\), large\(\infty\) softwares

- dynamic\(\infty\) computing: earth = computer = oxygen
Open Research Problem

Goal = automatic checking of bugs

Bugs = program runs unexpectedly
50-Year Achievements: in retrospect

revolved in 3 steps

• step 1) Definition of bugs (logic)

• step 2) Checking system (logic)

• step 3) Implementation (logic and computation)
Automatic Checking of Bugs: 1st gen.

syntax analysis: lexical analysis & parsing (70s)

- step 1) bug = program’s shape is wrong "\{intt x = 8*\}"

- step 2) Thm. “no bugs” ⇐⇒ correct shape

- step 3) Thm. “YES” ⇐⇒ “no bugs"
  - checking in \(\sim 10^4\) lines/sec
  - CFG languages
Automatic Checking of Bugs: 2nd gen.

Type checking/inference (90s, a pride of pgm’ng language area)

- step 1) bug = program’s execution is untypeful “free(x);”

- step 2) Thm. “no bugs” $\implies$ typeful exec.

- step 3) Thm. “YES” $\iff$ “no bugs”
  - checking/inferencing in $\sim 10^3$ lines/sec
  - HOT(higher-order & typed) languages v.s. C, C++, Java
Automatic Checking of Bugs: (3+k)th gen.

under way

- step 1) bug = program’s execution is not “as required”

- step 2) by program analysis/program logics/language technologies

- step 3) implementation
System Zoo is a tool for the generation-3 debugging technology (LET Project)
- use static analysis

- step 1) bug = program’s execution is not “as required”

- step 2) static analysis of programs against requirements

- step 3) implementation

- System Zoo automates step 2 and 3
Static Analysis

*a general technology for compile-time, automatic, and safe estimation of program's run-time properties*

- “general”: no limit on languages and properties
- “compile-time”: before execution
- “automatic”: program analyzes programs
- “safe”: result must subsume the reality
- “estimation”: cannot be exact in principle
Example: exception analysis

[Yi94, YiRy97, Yi98, YiRy02]

- bug = uncaught exceptions

- analysis = statically analyzing every possible uncaught exceptions

- requirement = the result must be the empty set
Example: KAIST SatRec’s Science Satellite (under way)

- bug = C module’s index variable is beyond [0,127]

- analysis = statically estimating index variable’s values

- requirement = the result must be within [0,127]
System Zoo

- a program analyzer generator
- a language for program properties/requirements

and ...
System Zoo

- to integrate with our nML compiler system (ropas.kaist.ac.kr/n)
  - a Korean dialect of Standard ML and OCaml: HOT family

- to transfer technology to the industry (int’l/domestic)
  - as “realistic/routine” as lex and yacc
Zoo Supports An Ensemble

- abstract interpretation

- conventional data flow analysis

- constraint-based analysis

- model checking
Use of Each Framework in Zoo

- variations in static analysis specification
  - abstract interpretation
  - data flow analysis
  - constraint-based analysis

- query about analysis result
  - model checking: computation-tree-logic (CTL) formula over analysis results
abstract interpretation
data flow analysis
constraint-based analysis

model checking

analysis query in Rabbit

L parser

L program

L program analyzer in nML

analysis query processor

analysis results

System Zoo

analysis specification for L programs in Rabbit

abstract interpretation
data flow analysis
constraint-based analysis
Talk Plan

1. Zoo’s viewpoint to program analysis

2. Rabbit: Zoo’s programming language

3. Unique issues
Program Analysis: Views from Zoo

Given a program

- phase 1: set-up equations
- phase 2: solve the equations
  - solution = graph \( \langle \text{abstract program states, flows} \rangle \)
- phase 3: make sense of the solution
  - checking properties = model checking
Input to Zoo

How to set-up equations: *abstract interpretation* style

\[ s \in \text{State} = \text{Var} \rightarrow \text{Sign} \]
\[ E \in \text{Expr} \times \text{State} \rightarrow \text{Sign} \times \text{State} \]

\[
E(x:=e,s) = \begin{align*}
& \text{let } (v_1,s_1) = E(e,s) \\
& \text{in} \ (v_1,s_1[v_1/x])
\end{align*}
\]

\[
E(e_1;e_2,s) = \begin{align*}
& \text{let } (v_1,s_1) = E(e_1,s) \\
& \quad (v_2,s_2) = E(e_2,s_1) \\
& \text{in} \ (v_2,s_2)
\end{align*}
\]

\[
E(e_1+e_2,s) = \begin{align*}
& \text{let } (v_1,s_1) = E(e_1,s) \\
& \quad (v_2,s_2) = E(e_2,s_1) \\
& \text{in} \ (\text{add}(v_1,v_2),s_2)
\end{align*}
\]

\[
E(\text{if } e_1 \ e_2 \ e_3, s) = \begin{align*}
& \text{let } (v_1,s_1) = E(e_1,s) \\
& \quad (v_2,s_2) = E(e_2,s_1) \\
& \quad (v_3,s_3) = E(e_3,s_1) \\
& \text{in} \ (v_2,s_2) \sqcup (v_3,s_3)
\end{align*}
\]
Correctness

Zoo users have to prove:

\[ \text{fix} F \xleftarrow{\alpha} \gamma \rightarrow \text{fix} \mathcal{F} \]

where

\[ \text{fix} F = [E] \quad \text{and} \quad \text{fix} \mathcal{F} = [E] \]

of

\[ F \in (Expr \times State \rightarrow Sign \times State) \rightarrow (Expr \times State \rightarrow Sign \times State) \]
\[ \mathcal{F} \in (Expr \times State \rightarrow Int \times State) \rightarrow (Expr \times State \rightarrow Int \times State) \]
\[ x := 1; \quad y := x + 1 \]

\[ X^\downarrow_i \in \text{State} \quad X^\uparrow_i \in \text{Sign} \times \text{State} \]

\[ X^\downarrow_0 = \bot \quad X^\uparrow_0 = X^\uparrow_2 \]

\[ X^\downarrow_1 = X^\uparrow_0 \quad X^\uparrow_1 = (X^\uparrow_{1a}.1, \quad X^\uparrow_{1a}.2[X^\uparrow_{1a}.1/x]) \]

\[ X^\downarrow_2 = X^\uparrow_{1a}.2 \quad X^\uparrow_2 = (X^\uparrow_{2a}.1, \quad X^\uparrow_{2a}.2[X^\uparrow_{2a}.1/y]) \]

\[ X^\downarrow_{2a} = X^\downarrow_2 \quad X^\uparrow_{2a} = (\text{add}(X^\downarrow_{2a}.2(x), 1), \quad X^\downarrow_{2a}.2) \]
Generated Analyzer Solves an Equation

\[
\begin{pmatrix}
X_1 \\
\vdots \\
X_n
\end{pmatrix}
= F
\begin{pmatrix}
X_1 \\
\vdots \\
X_n
\end{pmatrix}
\]

- The \( F \) is derived from the input Rabbit program

- Solution: \( \square\{\bot, F\bot, F^2\bot, \ldots\} \)
Solution: Fixpoint and Flow Graph

Fixpoint: equation solution \((X_i^\downarrow, X_i^\uparrow)\).

Flow graph:

\[
\begin{align*}
X_0^\uparrow & \leftarrow X_2^\uparrow \\
X_1^\downarrow & \leftarrow X_0^\downarrow \\
X_1^\uparrow & \leftarrow X_{1a}^\uparrow \\
X_2^\downarrow & \leftarrow X_{1.2}^\uparrow \\
X_2^\uparrow & \leftarrow X_{2a}^\uparrow \\
X_{2a}^\downarrow & \leftarrow X_2^\downarrow \\
X_{2a}^\uparrow & \leftarrow X_2^\uparrow
\end{align*}
\]
Generated Analyzer Answers to Query

- program behavior = analysis result, the flow graph

- query = Computation-Tree-Logic formula (a modal logic)
  - modality = \{A, E\} \times \{G, F, X, U\}
  - body = first-order predicate over \(X_i^{\downarrow}\) and \(X_i^{\uparrow}\)

Examples:

\[X_i^{\uparrow} \in Sign \times State\]
• Does variable $v$ remain positive?

$$AG(X^\uparrow(v) = \bigoplus)$$

• Can variable $v$ be positive?

$$EF(X^\uparrow(v) = \bigoplus)$$

• Does variable $v$ remain positive until $w$ is negative?

$$AU(X^\uparrow(v) = \bigoplus, X^\uparrow(w) = \bigoplus)$$

• From here, does variable $v$ remain positive?

$$v := x+y;$$
$$\text{## AG}(X^\uparrow.2(v)=\bigoplus)$$
$$\text{if } v > 0 \text{ then } v := v-2 \text{ else } v := v+1;$$
$$\ldots$$
All Inputs In Rabbit

Rabbit: a language for writing inputs to Zoo

- how-to-set-up equations in Rabbit: *abstract interpreters, data flow equations, constraints*

- what-to-query in Rabbit: CTL formula
Rabbit

- Type-inference: monomorphic typing, overloading, castings
  - primitive types $\exists$ user-defined sets/lattices
  - compound types $\exists$ tuple, sum, collection, function

- Module system
  - analysis module with/without a parameter analysis

- User-defined sets and lattices
- \{1\ldots10\}, \{a, b, c\}, 2^S, S_1 \times S_2, S_1 + S_2, S_1 \rightarrow S_2, \text{constraint set}
- S_\bot, 2^S, L_1 \times L_2, L_1 + L_2, S \rightarrow L, L_1 \rightarrow L_2, \text{set with an order}

- First-order functions
Rabbit Example

```
analysis TinyCfa =
    ana
        set Var = /Exp.var/
        set Lam = /Exp.expr/
        lattice Val = power Lam
        lattice State = Var -> Val

        widen Val with {/Lam(x,Lam _)/ ...} => top

        eqn E(/x/,s) = s(x)
            | E(/Lam(x,e)/, s) = {/Lam(x,e)/}
            | E(/App(e1,e2)/, s) = let val lams = E(/e1/, s)
                                val v = E(/e2/, s)
                                in
                                +{ E(e,s+bot[/x]=>v]) | /Lam(x,e)/ from lams }
        end
```
Rabbit Example

signature CFA = sig
  lattice Env
  lattice Fns = power /Ast.exp/
  eqn Lam: /Ast.exp/:index * Env -> Fns
end

analysis ExnAnal(Cfa: CFA) =
  ana
  set Exp = /Ast.exp/    set Var = /Ast.var/    set Exn = /Ast.exn/
  set UncaughtExns = power Exn
  constraint
    var = {X, P} index Var + Exp
    rhs = var
      | app_x(/Ast.exp/, var) | app_p(/Ast.exp/, var)
      | exn(Exn)             : atomic
      | minus(var, /Ast.exp/, power Exn) : atomic
      | cap(var, /Ast.exp/, Exn) : atomic
(* equation set-up rule *)

eqn Col /Ast.Var(x)/ = {}
| Col /Ast.Const/ = {}
| Col /Ast.Lam(x,e)/ = Col /e/
| Col /e as Ast.Fix(f,x,e',e'')/ = Col /e'/ + Col /e''/  
| Col /e as Ast.Case(e',k,e'',e''')/ = Col /e'/ + Col /e''/ + Col /e''''/  
| Col /e as Ast.Raise(e')/ = Col /e'/ + { P@e <- X@/e'/ }  
| Col /e as Ast.Handle(e', f as Ast.Lam(x,e''))/ = Col /e'/ + Col /e''/  
  + { X@/e/ <- X@/e'/, X@/e/ <- app_x(/f/, P@/e'/) }  
  + { X@/x/ <- P@/e'/, P@/e/ <- app_p(/f/, P@/e'/) }  

(* constraint closure rule *)
ccr  X@a <- app_x(/e/,X@b), /Ast.Lam(x,e’)/ in post Cfa.Lam@/e/
-----------------------------------------
    X@a <- X@/e’/, X@/x/ <- X@b

ccr  X@a <- app_x(/e/,P@b), /Ast.Lam(x,e’)/ in post Cfa.Lam@/e/
-----------------------------------------
    X@a <- X@/e’/, X@/x/ <- P@b

ccr  P@a <- app_p(/e/,X@b), /Ast.Lam(x,e’)/ in post Cfa.Lam@/e/
-----------------------------------------
    P@a <- P@/e’/, X@/x/ <- X@b

ccr  P@a <- app_p(/e/,P@b), /Ast.Lam(x,e’)/ in post Cfa.Lam@/e/
-----------------------------------------
    P@a <- P@/e’/, X@/x/ <- P@b

end
Issue I: Not a Blind Zoo

Zoo generates analyzers only when

- Rabbit exprs are monotonic or extensive: to guarantee termination of generated analyzers
- Rabbit exprs are typeful: well-formedness, efficiency
- Rabbit domains are lattices
- CTL formula are meaningful
Monotonicity and Extensionality Check

[MuYi02, YiEo02]

Static check of $F$

- so that $\sqcup\{ot, F\bot, F^2\bot, \cdots\}$ terminates

- monotonicity: $\forall \vec{X} \sqsubseteq \vec{Y}. F \vec{X} \sqsubseteq F \vec{Y}$

- extensionality: $\forall \vec{X}. \vec{X} \sqsubseteq F \vec{X}$
Some redundancies in:

\[ \sqcup \{ \bot, F \bot, F^2 \bot, \ldots \} \]

Differential algorithm with \( F' = \partial F / \partial \vec{X} \):

\[ \sqcup \{ \bot, F' \Delta_0, F' \Delta_1, \ldots \} \]

Achieved a linear scale-up of the algorithm cost
**Issue III: Rabbit’s Use in Fixpoint-Carrying Code**

FCC = a technology to check the safety of anonymous programs

<table>
<thead>
<tr>
<th>Code provider sends</th>
<th>Code consumer checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨code, $\vec{S}$⟩</td>
<td>- Scan(code) = ($\vec{X} = F\vec{X}$)</td>
</tr>
<tr>
<td></td>
<td>- Check($\vec{S} = F\vec{S}$)</td>
</tr>
</tbody>
</table>

Rabbit: the language for expressing the $F$ and $\vec{S}$.

- property to check = a Rabbit program
- Rabbit translation from for ML/C to for x86/MSIL/JVM
- System Zoo will be useful for building/checking safe programs
- System Zoo is not a toy: practice ↔ theory
- nML programming system is ready for the Zoo technology

May thy be proud of it, or not.
• Papers/details/concrete numbers/other works:
  ropas.kaist.ac.kr/~kwang/paper
  ropas.kaist.ac.kr/memo

• Softwares:
  - The Zoo/Rabbit definition: ropas.kaist.ac.kr/zoo
  - The nML definition/compiler: ropas.kaist.ac.kr/n

Thank you.