Who checks the typecheckers?

Bounded verification of type systems with symbolic execution

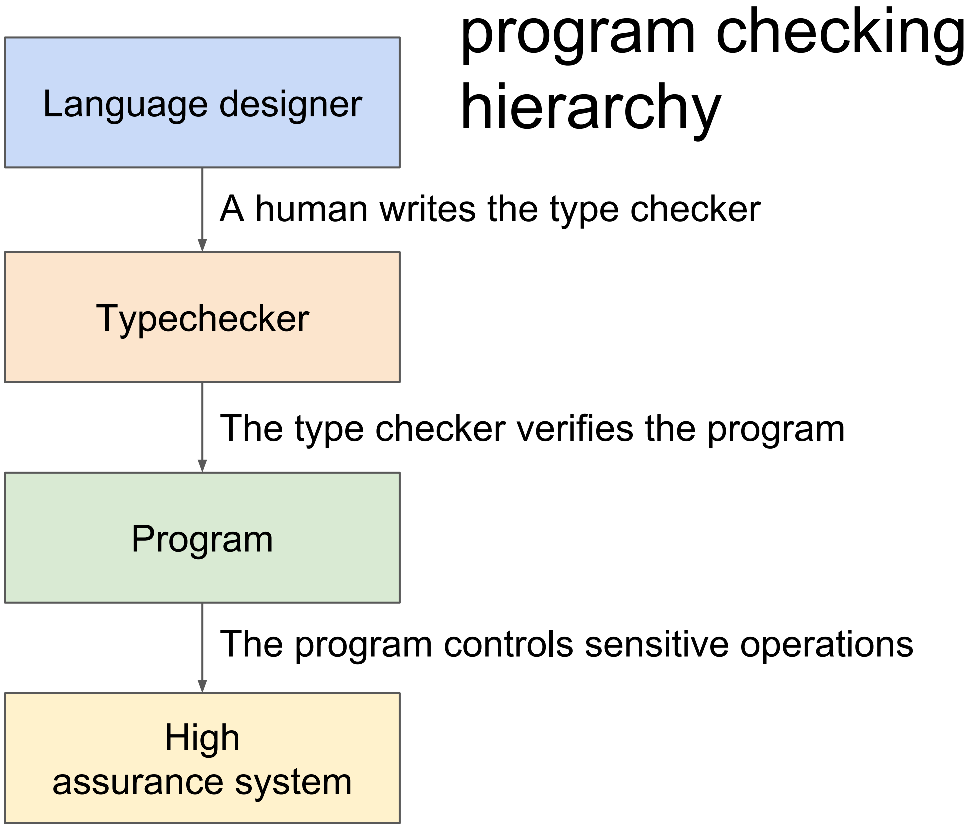
Kartik Chandra, Henry M. Gunn High School, Palo Alto
Rastislav Bodik, University of Washington, Seattle

PNW Programming Languages and Software Engineering Meeting
March 15, 2016

Today, type systems are not checked

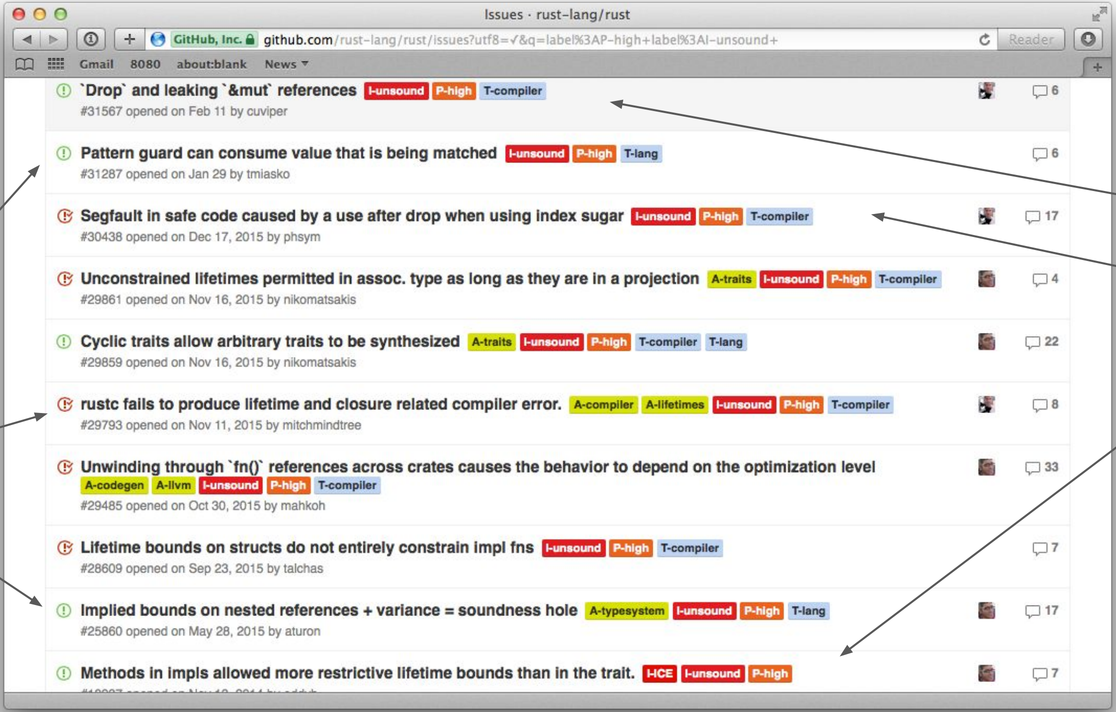
Most common programming languages such as Java, Scala, etc. do *not* have verified type systems.

Bugs in the typechecker could allow incorrect programs to be executed.



High-priority type safety issues (our long-term goal)

Memory freed by the compiler can be accessed later (security issue).



“Safe” code can spontaneously crash

Since Rusts’s typechecker has not been formally verified, there are many bugs.Here is a small selection of recently-reported **high-priority** type bugs in Rust, many of which have been fixed.

High-priority type safety issues (our long-term goal)

Ross Tate
14 mins · Ithaca, NY ·

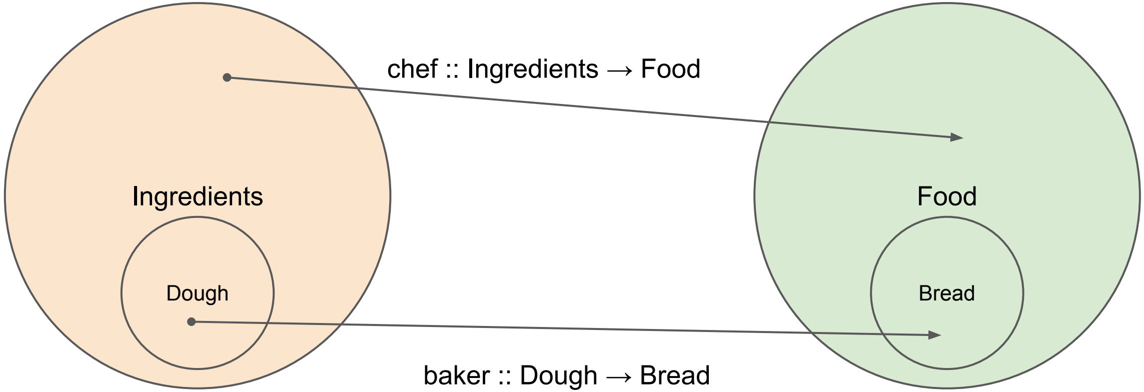
So, as an unexpected result of a fun conversation last week, Nada and I managed to prove Scala's type system unsound:

```
trait A { type L <: Nothing }
      trait B { type L >: Any}

      def toL(b: B)(x: Any) : b.L = x
      val p: B with A = null

      // we can create a value of type Nothing
      println(toL(p)("hello") : Nothing)

      // at runtime: java.lang.ClassCastException: java.lang.
String cannot be cast to scala.runtime.Nothing$
```

Motivating Example

```
class CHEF
feature
     cook(x: INGREDIENT): FOOD is …
End
class BAKER
feature
     cook(x: DOUGH): BREAD is
          DO knead(x); …; END
End
b : BAKER
c : CHEF
-- Eiffel allows this!
c := b
c.cook(cauliflower)
-- ERROR: cannot knead cauliflower
```

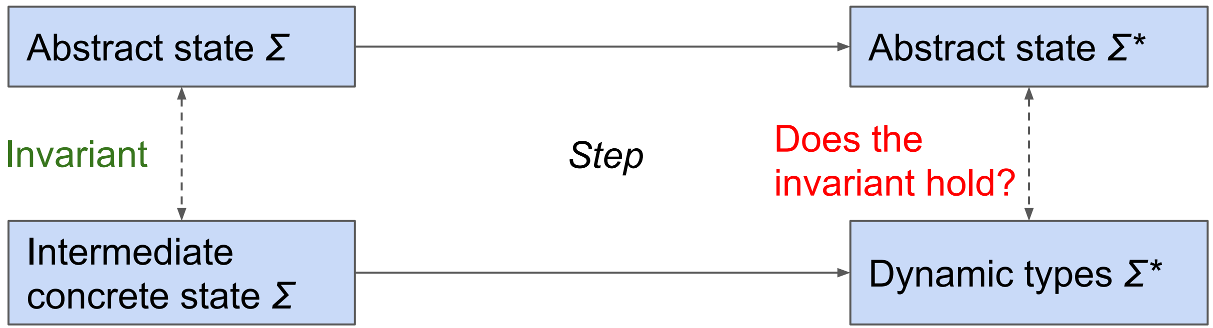
- Eiffel **allows** covariant method arguments!
- This is **unsafe** and leads to **incorrect** programs passing the typechecker.
- This is **hard to fix** because of legacy code breaking.



Existing model checking of type systems

For all intermediate states Σ
that **meet the invariant**…
For all possible **steps**…

(Explicit invariant, based on
Korat checking)

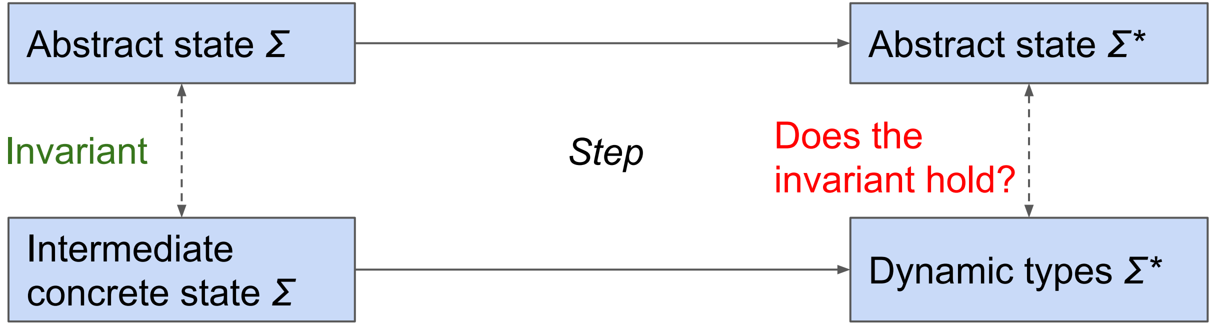


[Roberson *et al*, OOPSLA'08]

Existing model checking of type systems, vs. ours

For all intermediate states *Σ* that **meet the invariant**…
For all possible **steps**…

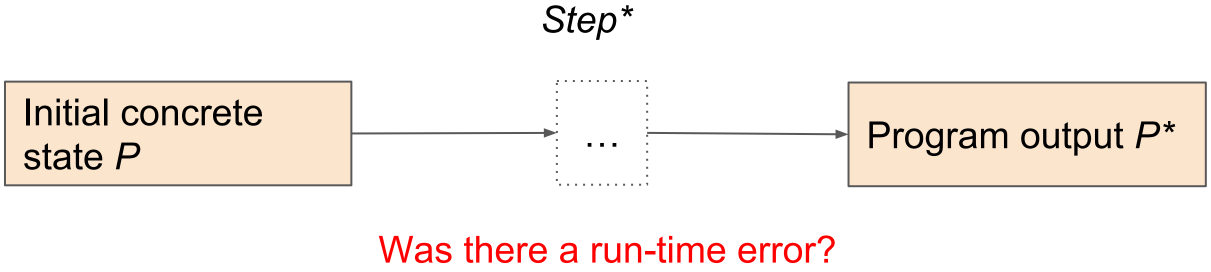
(Explicit invariant, based on Korat checking)



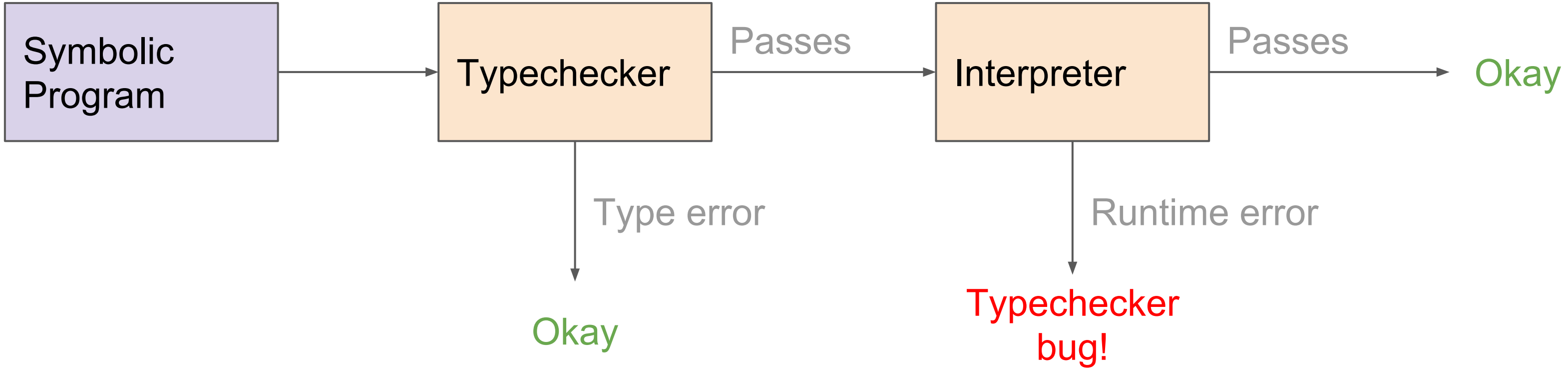
[Roberson *et al,* OOPSLA'08]

For all programs *P* that type check, does executing *P* go wrong?

(Without invariant, just run on the Rosette checker)



Our setup: symbolic typechecker checking for free



Rosette, a symbolic execution system for Racket, lets us run the typechecker-interpreter chain symbolically with very few changes.

Specifying the type system is easy

(From *Types and Programming Languages* by Benjamin Pierce.)

```
t ::=
      true
      false
      if t then t else t
      0
      succ t
      pred t
      iszero t
```

Typing rules for booleans — $\mathbb{B}$ **(typed)**

New syntactic forms

T ::= ... *(types...)*
  Bool *type of booleans*

New typing rules (t : T)

$$\texttt{true : Bool} \qquad \text{(T-True)}$$

$$\texttt{false : Bool} \qquad \text{(T-False)}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-If)}$$

Typing rules for numbers — $\mathbb{B}\ \mathbb{N}$ **(typed)**

New syntactic forms

T ::= ... *(types...)*
  Nat *type of natural numbers*

New typing rules (t : T)

$$\texttt{0 : Nat} \qquad \text{(T-Zero)}$$

Booleans — $\mathbb{B}$ **(untyped)**

Syntax

t ::= *(terms...)*
  true *constant true*
  false *constant false*
  if t then t else t *conditional*

v ::= *(values...)*
  true *value true*
  false *value false*

Evaluation ($t \longrightarrow t'$)

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-BoolBetaT)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-BoolBetaF)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-If)}$$

```
(define arithmetic-stx
  '((t ->
     | true | false
     | (if t t t)
     | zero
     | (succ t)
     | (pred t)
     | (iszero? t)))))
```

```
…
(define (type-if pred left right)
  (if (and
    (eq? (type-expr pred) 'bool)
    (eq? (type-expr left)
         (type-expr right)))
    (type-expr left)
    (assert #f)))
…
```

```
…
[(eq? tag 'if)
  (if (boolean?
    (runtime
      (tree-car (tree-cdr expr))))
  (if (runtime (tree-ref expr 1))
      (runtime (tree-ref expr 2))
      (runtime (tree-ref expr 3)))
  (assert #f "runtime error"))]
…
```

Bad typechecker

```
--snip--

(define (type-if pred left right)
  (if
    (eq? (type-expr pred) 'bool))
    (begin
       (type-expr left)
       (type-expr right))
    (assert #f)))

--snip--
```

Bad typechecker

```
--snip--

(define (type-if pred left right)
  (if
    (eq? (type-expr pred) 'bool))
    (begin
       (type-expr left)
       (type-expr right))
    (assert #f)))

--snip--
```

```
$ racket arithmetic.rkt
verify: counterexample found:

'(iszero?
   (if
     (iszero? (succ zero))
     (if (iszero? zero) zero zero)
     (iszero? zero)))
```

Fixed typechecker

```
--snip--

(define (type-if pred left right)
  (if (and
    (eq? (type-expr pred) 'bool)
    (eq? (type-expr left)
         (type-expr right)))
    (type-expr left)
    (assert #f)))

--snip--
```

Fixed typechecker

```
--snip--

(define (type-if pred left right)
  (if (and
    (eq? (type-expr pred) 'bool)
    (eq? (type-expr left)
         (type-expr right)))
    (type-expr left)
    (assert #f)))

--snip--
```

```
$ racket arithmetic.rkt
verify: no counterexample found
```

This does **not** mean it's necessarily error-free, just that no counterexample was found in the search space.

Talk overview

~~Model checking of type checkers~~

Symbolic programs

Assume statements

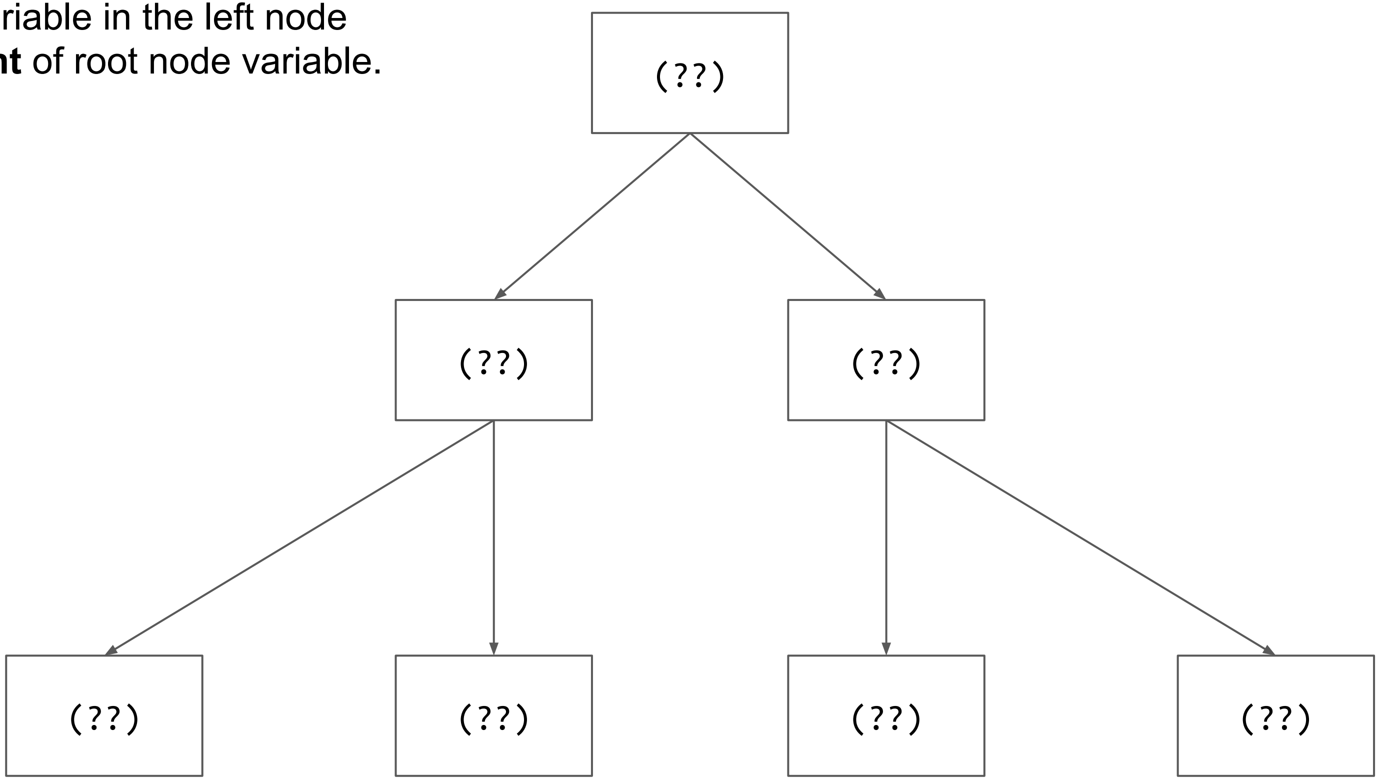
Featherweight Java Demo

Experiments

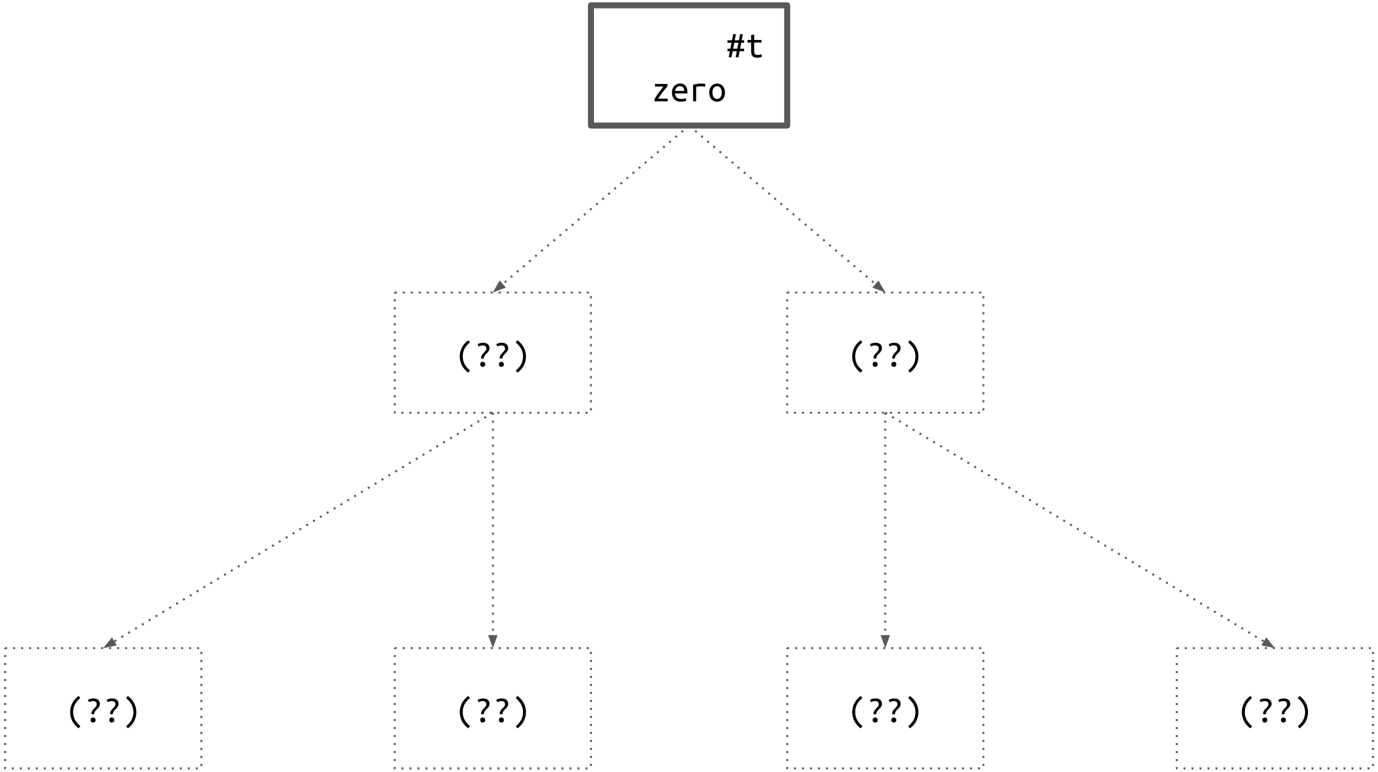
Remaining work

The binary encoding - symbolic domain

Symbolic variable in the left node **independent** of root node variable.

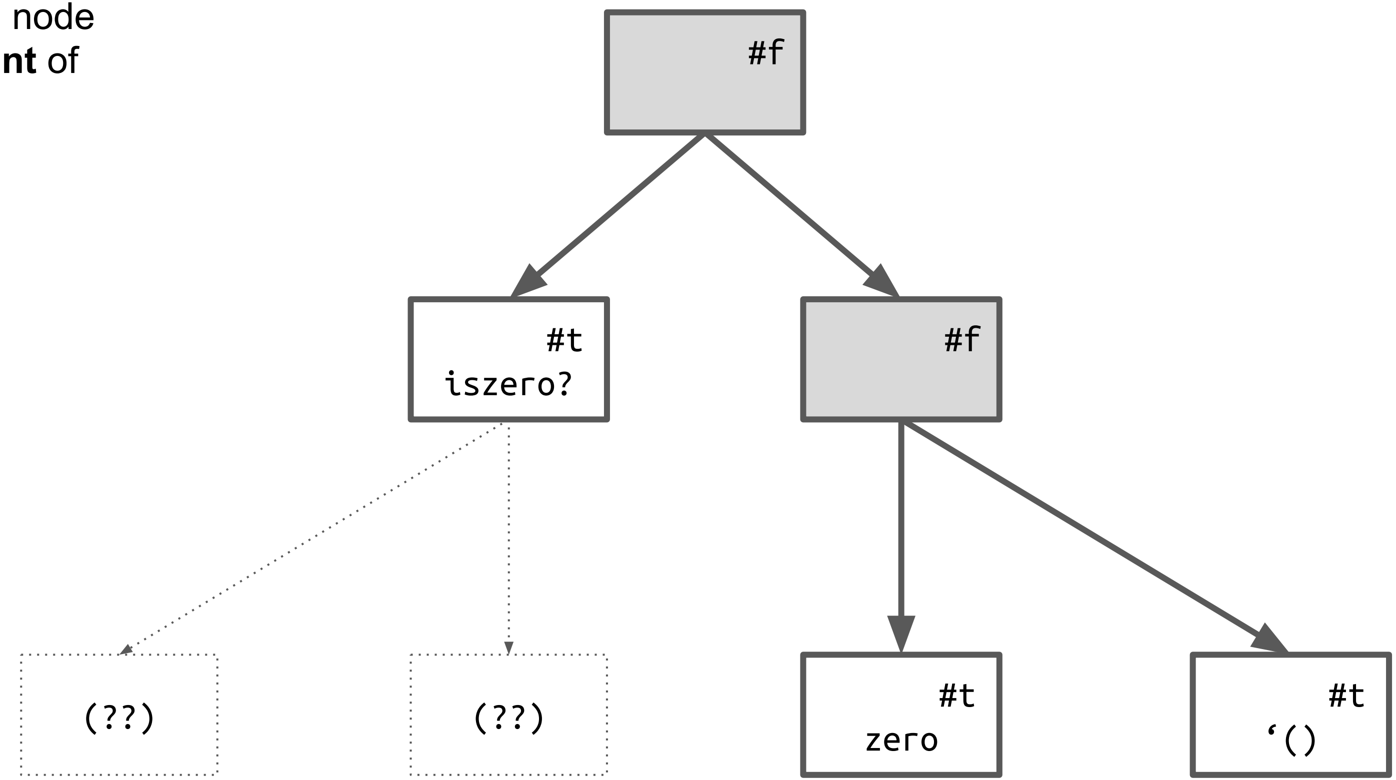


The binary encoding - `'zero`

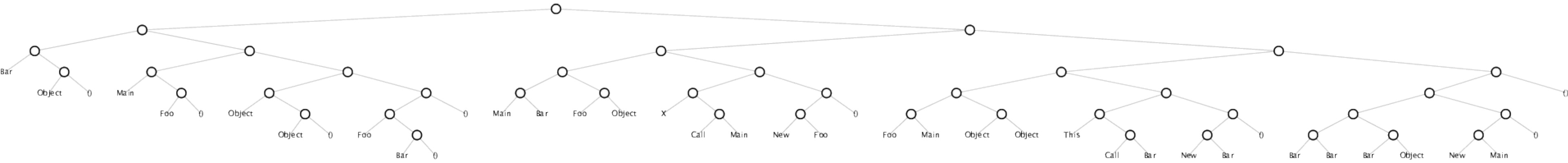


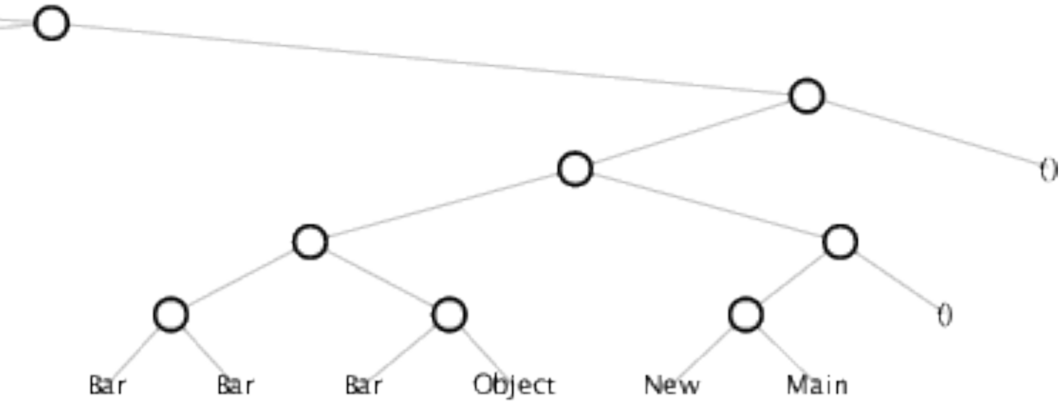
The binary encoding - ‘(iszero? zero)

Left branch node **independent** of root node.



A tree for Featherweight Java





Talk overview

~~**Model checking of type checkers**~~

~~**Symbolic programs**~~

Assume statements

Featherweight Java Demo

Experiments

Remaining work

What may happen during program evaluation

At runtime, a candidate program either

1. Terminates
2. Fails on run-time assertion (eg, null)
3. Diverges
4. Goes wrong (unchecked error) ← unsafe type system

We need to ignore 2 and 3. This requires assume statements which Rosette currently does not support.

Assumption by making the interpreter monadic

```
(define/rec (eval-expr x)
      […])
```

…becomes…

```
(define depth 10)
(define (eval-expr x)
      (assert (> depth 0))
      (unless (> depth 0) ‘depth-overflow)
      (set! depth (- depth 1))
      (define ans […])
      (set! depth (+ depth 1))
      ans)
```

```
(define (monadic f . args)
      (if (member ‘depth-overflow args)
          ‘depth-overflow
          (apply f args)))
```

Fun applications of typechecker checking

Are there programs accepted by a student's implementation but rejected by the reference implementation?

```
(verify
  #:assume    (student-checker   program)
  #:guarantee (reference-checker program))

→ (if (iszero? zero) zero (iszero? zero))
```

Do all safe programs typecheck? A two-line change!

```
(require “hindley-milner.rkt”)

(verify-correctness
   #:typechecker interpreter
   #:interpreter typechecker)

→ (lambda (a) b)
```

Demo! Featherweight Java

“Classic” covariance bug:

```
(assert
 (is-subclass? given-arg meth-arg))
(assert
 (is-subclass? given-out meth-out))
```

[Note: in “real” Java this overloads methods.]

Demo! Featherweight Java

```
Kartiks-MacBook-Pro:java kartikchandra$ time racket java.rkt
cpu time: 580 real time: 579 gc time: 25
#t
'((#<procedure:leaf-enum> leaf-enum? #<procedure:converter>) (leaf-enum Foo) . (leaf-enum ()))
Typecheck => Correct
Assumptions:    cpu time: 30156 real time: 30101 gc time: 2304
Guarantees:     cpu time: 7002 real time: 6988 gc time: 1452
✗ Verification failed.

Counterexample:
'(((Bar Object) (Main Foo) (Object Object) (Foo Bar))
  (((Main . Bar) Foo . Object) (X Call . Main) (New . Foo))
  (((Foo . Main) Object . Object) (This Call . Bar) (New . Bar))
  (((Bar . Bar) Bar . Object) (New . Main)))
Visualization in ./counterexample.pdf

Assumption:
#t

Guarantee:
assert: Method not found.
  context...:
   /Users/kartikchandra/Desktop/types/tree/java/bounded.rkt:62:16
   /Users/kartikchandra/Desktop/types/tree/java/bounded.rkt:62:16
   /Users/kartikchandra/Desktop/types/tree/java/java.rkt:161:8
   /Users/kartikchandra/Desktop/types/tree/java/tree-verify.rkt:26:12: loop
   /Users/kartikchandra/Desktop/types/tree/java/java.rkt: [running body]

real    3m53.383s
user    0m49.453s
sys     0m1.739s
Kartiks-MacBook-Pro:java kartikchandra$
```

Demo! Featherweight Java

“Classic” covariance bug:

```
(assert
 (is-subclass? given-arg meth-arg))
(assert
 (is-subclass? given-out meth-out))
```

[Note: in “real” Java this overloads methods.]

```
$ racket java.rkt
Counterexample found.
```

```
class Bar extends Object {
      public Object beep(Bar X) {
            return new Main();
      } }
class Foo extends Bar {
      public Object main(Object X) {
            return this.beep(new Bar());
      } }
class Main extends Foo {
      public Object beep(Foo X) {
            return X.main(new Foo());
      } }

// new Main().main(new Object())
```

We can explore same space with Rosette encoding

	Roberson *et al*		**Chandra**	
Languages	**Program size**	**Time**	**Program size**	**Time**
Arithmetic	Up to 3281 AST nodes	38s	Up to 2047 AST nodes	15s
FJ	Up to 341 AST nodes	475s	Up to 511 AST nodes	190s

Our Results

Possible to check a type system “for free” via symbolic execution

- Using just an existing typechecker and interpreter (as opposed to single-step transition)
- Using Rosette for symbolic execution (as opposed to an explicit Korat algorithm)

Two main tricks are required to make this work:

- Efficient symbolic encoding of AST data structures
- “Monadic” macros to provide assume in Rosette and bound infinite loops

Remaining work:

- Encode remaining languages from Roberson (IMP, Mini Java, Ownership Java)
- Encoding modern type systems such as Scala’s

Thanks!

Kartik Chandra, Henry M. Gunn High School
Rastislav Bodik, University of Washington