Symbolic Types for Lenient Symbolic Execution

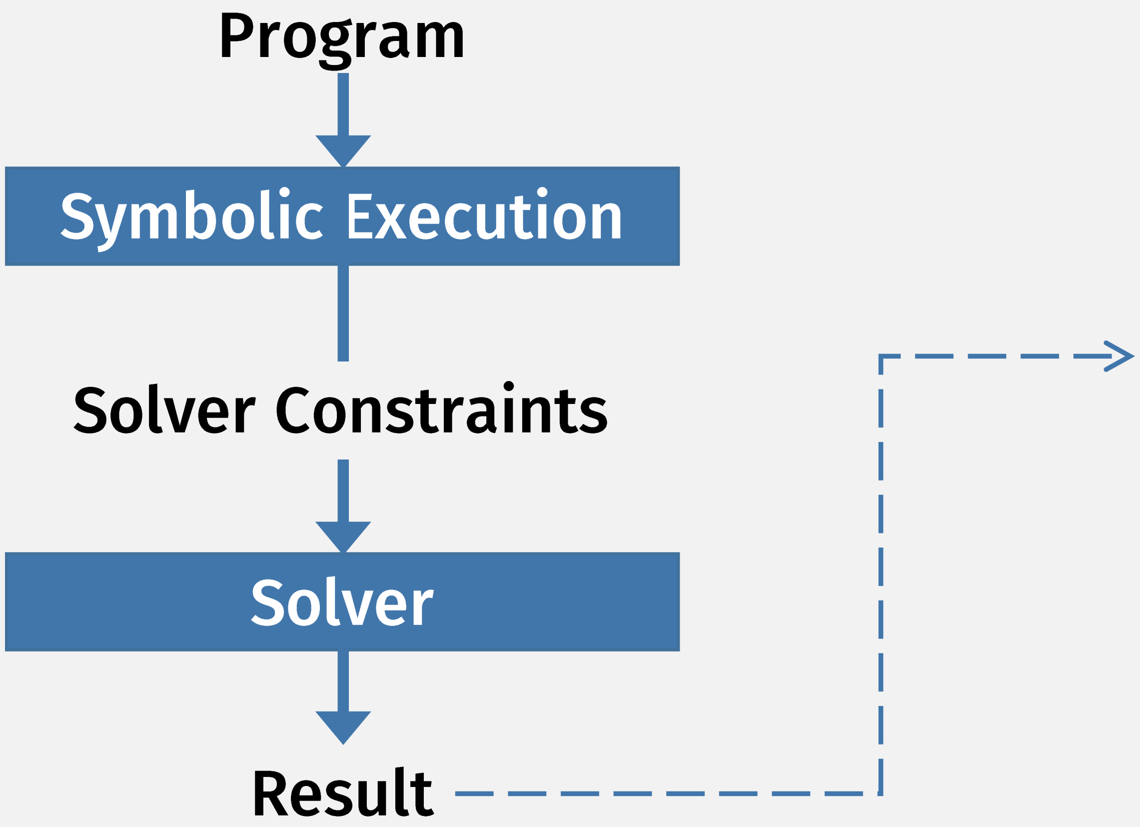
Stephen Chang, Alex Knauth

Northeastern University

Emina Torlak

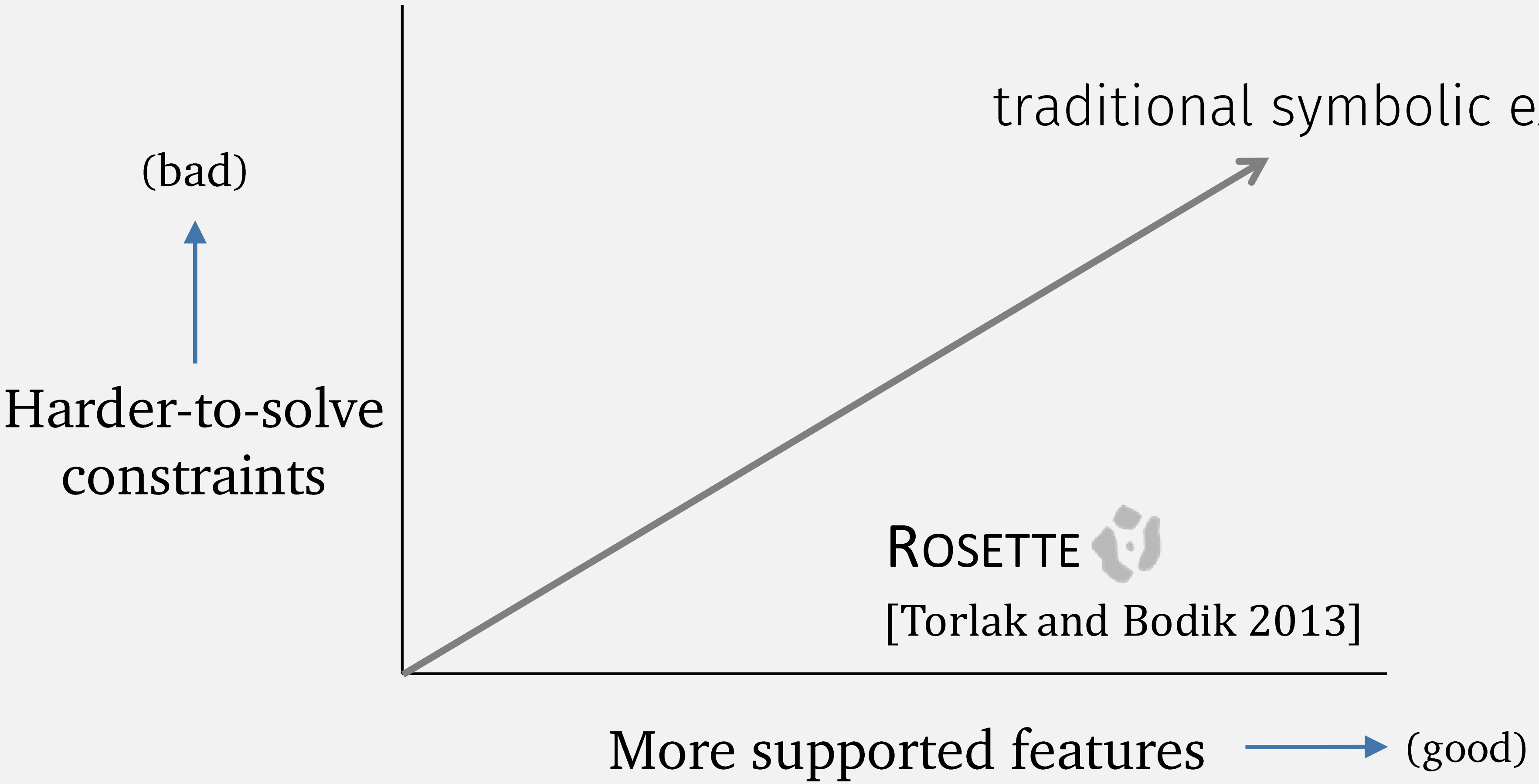
University of Washington

Solver-aided programming is on the rise



- Testing
- Verification
- Synthesis

More features vs constraint complexity



Rosette

- Simple constraints

- Supports many features

Applications

Bagpipe	A language for specifying BGP policies and verifying that an Internet Service Provider's router configurations implement these policies.
Chlorophyll	A synthesis-aided programming model and compiler for GreenArrays GA144, a minimalist low-power spatial architecture.
Cosette	A framework for reasoning about SQL equivalences.
Ferrite	A framework for specifying and checking file system crash-consistency models.
Greenthumb	A framework for constructing superoptimizers.
MemSynth	A language and tool for verifying, synthesizing, and disambiguating memory consistency models.
Neutrons	A verifier for a subset of EPICS. Currently in use at the University of Washington Clinical Neutron Therapy System.
Synapse	A framework for specifying and solving optimal synthesis problems.
Ocelot	An engine for solving, verifying, and synthesizing specifications in bounded relational logic.
Wallingford	An experimental constraint reactive programming language.
More	Demo languages and tools for secure stack machines, data-parallel programing, and web-scraping.

https://emina.github.io/rosette/apps.html

Rosette

- Simple constraints →
- Lift small language subset (Racket) for symbolic execution

"lenient symbolic execution"

- Supports many features →
- Macro-express more complex features (solver-aided DSLs)
- Allow interleaving unlifted features (data structures)

Rosette

- Simple constraints → Lift small language subset (Racket) for symbolic execution

"lenient symbolic execution"

- Supports many features → Macro-express more

Problem:
Manual management of symbolic values

- Allow interleaving unlifted features (data structures)

Our contribution: Typed Rosette

- Simple constraints →
- Lift small language subset (Racket) for symbolic execution

“lenient symbolic execution”

- Supports many features →
- Macro-express more

unlifted

ctures)

~~Problem~~ Solution:

~~Manual management of symbolic values~~

Types manage symbolic values

Example

```
#lang racket

(define (my-new-sorting-algo lst) ....)

(test (my-new-sorting-algo (list 1 2 3)))
(test (my-new-sorting-algo (list 1 3 2)))
....
```

Example

```
#lang rosette

(define (my-new-sorting-algo lst) ....)

(test (my-new-sorting-algo (list 1 2 3)))
(test (my-new-sorting-algo (list 1 3 2)))
....

(define-symbolic ^x ^y ^z integer?)
(verify
 (assert (sorted?
  (my-new-sorting-algo (list ^x ^y ^z)))))
; SOLVER: ✓ (any 3-element list is sorted correctly)
```

Correctness specification

```
#lang rosette

; ∀i,j: i < j => lst[i] ≤ lst[j]
(define (sorted? lst)
  (define-symbolic ^i ^j integer?)
  (implies (< ^i ^j)
           (<= (list-ref lst ^i)
               (list-ref lst ^j))))
```

Sort function

```
#lang rosette

(define (my-sort lst)
  (if (null? lst)
      lst
      (let loop ([x1 (car lst)]
                 [rst1 (my-sort (cdr lst))])
        (if (null? rst1)
            (list x1)
            (let ([x2 (car rst1)]
                  [rst2 (cdr rst1)])
              (if (< x1 x2)
                  (cons x1 rst1)
                  (cons x2 (loop x1 rst2))))))))
```

Sorting, with pattern match (success)

```
#lang rosette

(require my-pattern-match-lib) ; allowed by “lenient” symb exe

(define my-sort $null        = null
        my-sort ($: x xs) = (helper x (my-sort xs)))

(define helper x $null        = (list x)
        helper x ($: y ys) @ (< x y)     = (: x y ys)
                           @ otherwise = (: y (helper x ys)))

(verify (sorted? (my-sort (list ^x ^y ^z)))) ; SOLVER: ✓
```

Sorting, with list lib (fail)

```
#lang rosette

(require my-pattern-match-lib list-lib)

(define my-sort $null       = null
        my-sort ($: x xs) = (insert x (my-sort xs)))




(verify (sorted? (my-sort (list ^x ^y ^z)))) ; SOLVER: ✗
; counterexample: ^i = 0, ^j = 1, ^x = 1, ^y = -16, ^z = 0
```

Sorting, with list lib (unknown fail)

```
#lang rosette

(require my-pattern-match-lib list-lib)

(define my-sort $null       = null
        my-sort ($: x xs) = (insert x (my-sort xs)))
```

Problem: given symbolic, but must be concrete

```
(verify (sorted? (my-sort (list ^x ^y ^z)))) ; SOLVER: ✗
; counterexample: ^i = 0, ^j = 1, ^x = 2, ^y = -16, ^z = 0

; But the counterexample sorts correctly ?????
(my-sort (list 1 -16 0)) ; => (list -16 0 2)
```

Sorting, with types (fail with type msg)

```
#lang typed/rosette

(require my-pattern-match-lib typed-list-lib)

(define my-sort $null      = null
        my-sort ($: x xs) = (insert x (my-sort xs)))
; TYPE ERR: `insert` 1st arg is symbolic, expected concrete Int
```

A symbolic λ-calculus

$$e ::= n \mid x \mid \lambda x\!:\tau.\, e \mid e\ e \mid \mathrm{add1} \mid \mathit{if}\ e\ e\ e \mid \mathit{set!}\ x\ e \mid \boxed{\hat{x}^{\tau}} \mid \ \ldots$$

$$\tau ::= \mathrm{Int} \mid \tau \to \tau \mid \boxed{\hat{\tau}} \mid \ldots$$

T-SymInt

Safe, but is insufficient for lenient symbolic execution.

<u>Where</u>: Int <: $\widehat{\mathrm{Int}}$,

<u>eg</u>: $5 : \mathrm{Int}$ <u>and</u> $5 : \widehat{\mathrm{Int}}$

What should be the type of add1?

$$\text{add1} : \widehat{\text{Int}} \to \widehat{\text{Int}}$$

$$\text{add1 } 5 : \widehat{\text{Int}}$$

“Symbolicness” should not spread too easily.

- Symbolic type
- Cannot be used with concrete functions
- I.e., cannot be used with lenient symbolic execution

Our type system goals

Safe:

- Symbolic values do not flow to unsupported positions.

(see theorem in paper)

Useful:

- For programs using lenient symbolic execution,
- Concrete types are preserved as much as possible.

"concreteness polymorphism"
(this talk)

Concreteness polymorphism

1. Function intersection types (this talk)

2. Path concreteness markers (this talk)

3. Union types and occurrence typing

Function intersection types

$$\tau = \mathrm{Int} \mid \tau_{fn} \mid \tau_{fn} \cap \tau_{fn} \mid \hat{\tau} \mid \ldots$$

$$\tau_{fn} = \tau \to \tau$$

$$\mathrm{add1} : \boxed{\mathrm{Int} \to \mathrm{Int}} \cap \widehat{\mathrm{Int}} \to \widehat{\mathrm{Int}}$$

Sub-∩-1

$$\frac{\tau_{fn_1} <: \tau_{fn}}{\boxed{\tau_{fn_1}} \cap \tau_{fn_2} <: \tau_{fn}}$$

Sub-∩-2

$$\frac{\tau_{fn_2} <: \tau_{fn}}{\tau_{fn_1} \cap \tau_{fn_2} <: \tau_{fn}}$$

$$\mathrm{add1}\ 5 : \mathrm{Int}$$

Function intersection types

$$\tau = \mathrm{Int} \mid \tau_{fn} \mid \tau_{fn} \cap \tau_{fn} \mid \hat{\tau} \mid \ldots$$

$$\tau_{fn} = \tau \to \tau$$

$$\mathrm{add1} : \mathrm{Int} \to \mathrm{Int} \cap \boxed{\widehat{\mathrm{Int}} \to \widehat{\mathrm{Int}}}$$

Sub-∩-1

$$\frac{\tau_{fn_1} <: \tau_{fn}}{\tau_{fn_1} \cap \tau_{fn_2} <: \tau_{fn}}$$

Sub-∩-2

$$\frac{\tau_{fn_2} <: \tau_{fn}}{\tau_{fn_1} \cap \boxed{\tau_{fn_2}} <: \tau_{fn}}$$

$$\mathrm{add1}\ 5 : \mathrm{Int}$$

$$\mathrm{add1}\ \hat{x}^{\mathrm{Int}} : \widehat{\mathrm{Int}}$$

Concreteness polymorphism

1. Function intersection types (this talk)
2. $\boxed{\text{Path concreteness markers}}$ (this talk)
3. Union types and occurrence typing

Path concreteness

```
#lang typed/rosette

(define x 0)      ; x = 0, concrete value with concrete type
```

Path concreteness

```
#lang typed/rosette

(define x 0)     ; x = 0, concrete value with concrete type
(set! x 3) ; SAFE: x = 3, concrete value with concrete type
```

Path concreteness

```
#lang typed/rosette

(define x 0)     ; x = 0, concrete value with concrete type
(set! x 3) ; SAFE: x = 3, concrete value with concrete type

(define-symbolic ^b boolean?)

(if ^b ; symbolic test, so both paths executed
    (set! x 10)
    (set! x 11))
```

Path concreteness

```
#lang typed/rosette

(define x 0)       ; x = 0, concrete value with concrete type
ALLOW (set! x 3) ; SAFE: x = 3, concrete value with concrete type

(define-sym

(if ^b ; sy                                  ed
  REJECT (set! x 10)    ; UNSAFE
  REJECT (set! x 11))   ; UNSAFE

; result:  x is symbolic value (that is either 10 or 11)
; but still has concrete type
```

For safe and useful mutation, track path concreteness.

Tracking path concreteness

$\pi = \bullet \mid \circ$

$\bullet$ = concrete path

$\circ$ = possibly symbolic path

Tracking path concreteness

$\pi = \bullet \mid \circ$

$\bullet$ = concrete path

$\circ$ = possibly symbolic path

<u>Type-checking rules depend on path concreteness:</u>

T-IF-CONC$^{\bullet}$

$$\frac{\begin{array}{c} \Gamma \vdash^{\bullet} e_1 : \tau_1 \\ \mathit{concrete?}\,\tau_1 \\ \Gamma \boxed{\vdash^{\bullet}} e_2 : \tau \\ \Gamma \boxed{\vdash^{\bullet}} e_3 : \tau \end{array}}{\Gamma \vdash^{\bullet} \mathtt{if}\ e_1\ e_2\ e_3 : \tau}$$

Concrete test: no path change

T-IF-SYM$^{\bullet}$

$$\frac{\begin{array}{c} \Gamma \vdash^{\bullet} e_1 : \tau_1 \\ \mathit{symbolic?}\,\tau_1 \\ \Gamma \boxed{\vdash^{\circ}} e_2 : \tau \\ \Gamma \boxed{\vdash^{\circ}} e_3 : \tau \end{array}}{\Gamma \vdash^{\bullet} \mathtt{if}\ e_1\ e_2\ e_3 : \widehat{\tau}}$$

Symbolic test: change path to symbolic

Safe mutation

Concrete path: all types ok

$$\textsc{T-Set!}^{\bullet}\quad \frac{\boxed{x:\tau} \in \Gamma \qquad \Gamma \overset{\bullet}{\vdash} e : \tau}{\Gamma \overset{\bullet}{\vdash} \texttt{set!}\ x\ e : \texttt{Unit}}$$

Symbolic path: requires symbolic type

$$\textsc{T-Set!}^{\circ}\quad \frac{\boxed{x:\widehat{\tau}} \in \Gamma \qquad \Gamma \overset{\circ}{\vdash} e : \widehat{\tau}}{\Gamma \overset{\circ}{\vdash} \texttt{set!}\ x\ e : \texttt{Unit}}$$

Path concreteness and function definitions

```
#lang typed/rosette

(define x 0) ; x = 0, concrete value with concrete type

(define (f [y : Int]) (set! x y)) ; SAFE?
```

Path concreteness and function definitions

```
#lang typed/rosette

(define x 0) ; x = 0, concrete value with concrete type

(define (f [y : Int]) (set! x y))

(f 1) ; SAFE: x = 1, concrete value with concrete type
```

Path concreteness and function definitions

```
#lang typed/rosette

(define x 0) ; x = 0, concrete value with concrete type

(define (f [v : Int]) (set! x v))
```

ALLOW

```
(f 1) ; SAFE                                              type
```

Must consider path concreteness
at each function call site.

```
(define-symbolic ^b boolean?)
(if ^b (f 2) (f 3)) ; UNSAFE
```

REJECT REJECT

```
; x = (ite ^b 2 3), symbolic value with concrete type
```

Path concreteness and functions

$$\tau_{fn} ::= \pi : \tau \to \tau$$

Path concreteness of function type and calling context must match:

$$\frac{\Gamma \vDash^{\pi} e_1 : \boxed{\pi} : \tau_1 \to \tau_2 \qquad \Gamma \vDash^{\pi} e_2 : \tau_1}{\Gamma \boxed{\vDash^{\pi}} e_1\, e_2 : \tau_2}\ \textsc{T-App}^{\pi}$$

Path concreteness and function definitions

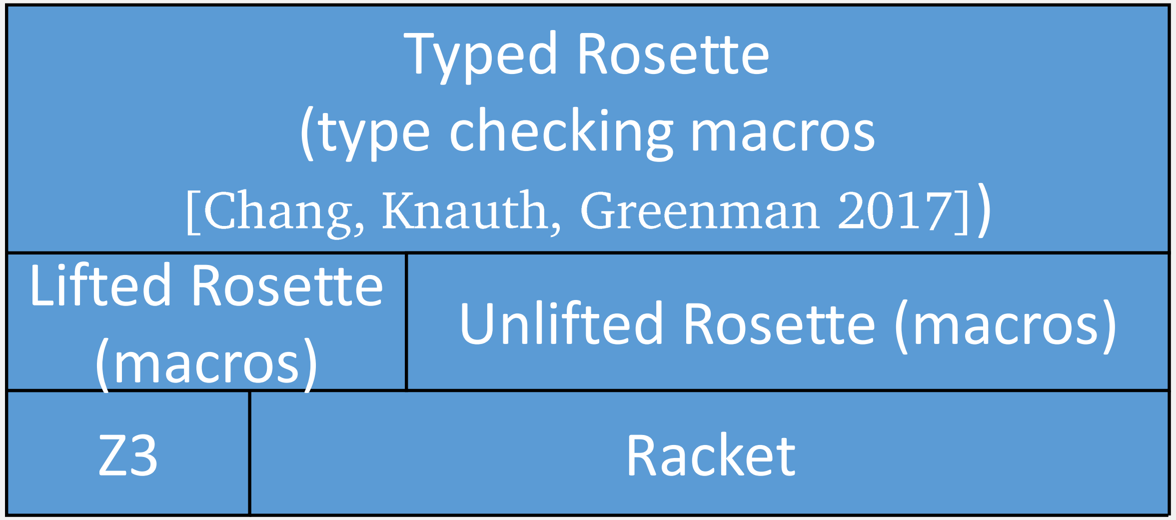
$$\tau_{fn} ::= \pi : \tau \to \tau$$

$$\textsc{T-Lam-ConcPath}^{\pi}\ \frac{\Gamma, x:\tau \boxed{\vdash^{\bullet}} e \ : \ \tau'}{\Gamma \vdash^{\pi} \lambda x:\tau\,.\,e \ : \boxed{\bullet}:\tau \to \tau'}$$

$$\textsc{T-Lam-SymPath}^{\pi}\ \frac{\Gamma, x:\tau \boxed{\vdash^{\circ}} e \ : \ \tau'}{\Gamma \vdash^{\pi} \lambda x:\tau\,.\,e \ : \boxed{\circ}:\tau \to \tau'}$$

Check body twice, with concrete and symbolic path

Typed Rosette implementation



Type checking macros

```
(define-typerule set!
 [(set! x e) ≫ #:when (sym-path?)
  [⊢ x ≫ x- ⇒ τ]
  #:fail-unless (symbolic? τ)
  "sym path requires sym type“
  [⊢ e ≫ e- ⇐ τ]
  ------------------------------
  [⊢ (rosette:set! x- e-) ⇒ Unit]]
 [(set! x e) ≫ #:when (conc-path?)
  [⊢ x ≫ x- ⇒ τ]
  [⊢ e ≫ e- ⇐ τ]
  ------------------------------
  [⊢ (rosette:set! x- e-) ⇒ Unit])
```

T-SET!$^{\circ}$

$$\frac{x : \widehat{\tau} \in \Gamma \qquad \Gamma \overset{\circ}{\vdash} e : \widehat{\tau}}{\Gamma \overset{\circ}{\vdash} \mathsf{set!}\ x\ e : \mathsf{Unit}}$$

T-SET!$^{\bullet}$

$$\frac{x : \tau \in \Gamma \qquad \Gamma \overset{\bullet}{\vdash} e : \tau}{\Gamma \overset{\bullet}{\vdash} \mathsf{set!}\ x\ e : \mathsf{Unit}}$$

Evaluation

Ported ~10000 loc from Rosette, with ~6000 loc tests

Name	Untyped LoC	+Typed LoC
basic		
fsm	162	+86
bv	434	+101
ifc	962	+137
synthcl	2632	+615
ocelot	1757	+396
inc	5445	+634

New typed code includes:

- Adding type annotations for functions
- Implementing new type rules for syntax extensions
- Casts to help the type checker

Takeaway

- Rosette's "lenient" symbolic execution:
  - Avoids hard-to-solve constraints,
  - Allows writing full-featured, solved-aided programs,
  - But can be hard to debug.
- Typed Rosette:
  - Safe:
    - Ensures symbolic values do not flow to unsupported locations,
  - Useful:
    - For writing lenient symbolic execution programs,
    - Preserves concreteness in types with concreteness polymorphism.

https://github.com/stchang/typed-rosette

(requires Racket)

```
raco pkg install --auto typed-rosette
```