Solver-aided programming: getting started

Emina Torlak
University of Washington
emina@cs.washington.edu
homes.cs.washington.edu/~emina/
Solver-aided programming in two parts: (1) getting started and (2) going pro
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:** (1) getting started and (2) going pro
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
(1) **getting started** and (2) **going pro**

How to use a solver-aided language: the workflow, constructs, and gotchas.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
(1) **getting started** and (2) **going pro**

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
(1) **getting started** and (2) **going pro**

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**

(1) **getting started** and (2) **going pro**

How to use a solver-aided language: the *workflow*, constructs and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Classic programming: from spec to code

specification

P(x) {
  ...
  ...
}

Classic programming: check code against spec

check the specification on concrete inputs

P(x) {
  ...
  ...
} assert safe(2, P(2))
Solver-aided programming: add symbolic values

check the specification on symbolic inputs

\[
P(x) \{
  ...
  ...
}
assert \text{safe}(x, P(x))
\]

The symbolic value \( x \) stands for an arbitrary integer.
Solver-aided programming: query code against spec

The symbolic value \( x \) stands for an arbitrary integer. The runtime uses the solver to determine the concrete meaning of \( x \) in response to solver-aided queries.
Solver-aided programming: \textit{query} code against spec

$$P(x) \{ ... \}$$

assert safe($x$, $P(x)$)

queries

verify debug solve synthesize

solver-aided language

SMT solver
Solver-aided programming: verify code against spec

Find an input on which the program fails.

\[ \exists x . \neg \text{safe}(x, P(x)) \]

\[
P(x) \{
\ldots
\ldots
\}
assert \text{safe}(x, P(x))
\]
Solver-aided programming: *debug* code against spec

Find an input on which the program fails. Localize bad parts of the program.

```
P(x) { 
    v = x + 2
    ... 
} 
assert safe(x, P(x))
```

```
∃ x . ¬ safe(x, P(x))
x = 42 ∧ safe(x, P(x))
```
Solver-aided programming: solve for values from spec

- Find an input on which the program fails.
- Localize bad parts of the program.
- Find values that repair the failing run.

P(x) {
  v = choice()
  ...
}
assert safe(x, P(x))

∃ x . ¬ safe(x, P(x))

SMT solver

solver-aided
language

v = choice()

42
40
Solver-aided programming: *synthesize* code from spec

Find an input on which the program fails.
Localize bad parts of the program.
Find values that repair the failing run.
Find code that repairs the program.

**verify**  
**debug**  
**solve**  
**synthesize**

\[ P(x) \{ \]
\[ v = ?? \]
\[ \ldots \]
\[ \} \]
\[ \text{assert safe}(x, P(x)) \]

**solver-aided language**  
**SMT solver**

\[ \exists x . \neg \text{safe}(x, P(x)) \]
\[ x = 42 \land \text{safe}(x, P(x)) \]
\[ \exists v . \text{safe}(42, P_v(42)) \]
\[ \exists e . \forall x . \text{safe}(x, P_e(x)) \]
Use **assertions** and **symbolic values** to express the specification.

Ask **queries** about program behavior (on arbitrary inputs) with respect to the specification.

\[ \exists x . \neg \text{safe}(x, P(x)) \]
\[ x = 42 \land \text{safe}(x, P(x)) \]
\[ \forall v . \text{safe}(42, P_v(42)) \]
\[ \exists e. \forall x. \text{safe}(x, P_e(x)) \]
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

Solver-aided programming in two parts: (1) **getting started** and (2) going pro

How to use a solver-aided language: the workflow, **constructs**, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Rosette extends Racket with solver-aided constructs

= +

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
Rosette extends **Racket** with solver-aided constructs

“A programming language for creating new programming languages”

A modern descendent of Scheme and Lisp with powerful macro-based meta programming.

- `(define-symbolic id type)`
- `(define-symbolic* id type)`
- `(assert expr)`
- `(verify expr)`
- `(debug [type ...+] expr)`
- `(solve expr)`
- `(synthesize #:forall expr #:guarantee expr)`

**symbolic values**

**assertions**

**queries**
Rosette extends **Racket** with solver-aided constructs

```racket
#lang rosette
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
#:forall expr
#:guarantee expr)
```
define-symbolic creates a fresh symbolic constant of the given type and binds it to the variable id.
**Rosette constructs: define-symbolic**

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

**define-symbolic** creates a fresh symbolic constant of the given type and binds it to the variable \textit{id}.

> (define-symbolic x integer?)
Rosette constructs: define-symbolic

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

\[(\text{define-symbolic } \text{id} \text{ type})\]
\[(\text{define-symbolic* id type})\]
\[(\text{assert expr})\]
\[(\text{verify expr})\]
\[(\text{debug [type ...+] expr})\]
\[(\text{solve expr})\]
\[(\text{synthesize #:forall expr #:guarantee expr})\]

**define-symbolic** creates a fresh symbolic constant of the given type and binds it to the variable **id**.

> (define-symbolic x integer?)
> (+ 1 x 2 3)
> (+ 6 x)

Symbolic values of a given type can be used just like concrete values of that type.
Rosette constructs: define-symbolic

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

**define-symbolic** creates a fresh symbolic constant of the given type and binds it to the variable id.

> (define (same-x)
  (define-symbolic x integer?)
  x)
> (same-x)
  x
> (same-x)
  x
> (eq? (same-x) (same-x))
  #t

id is bound to the same constant every time **define-symbolic** is evaluated.

Symbolic values of a given type can be used just like concrete values of that type.
**Rosette constructs: define-symbolic**

A type that is efficiently supported by SMT solvers: booleans, integers, reals, bitvectors, uninterpreted functions.

\[
(\text{define-symbolic } \text{id} \text{ type})
\]

\[
(\text{define-symbolic* } \text{id} \text{ type})
\]

\[
(\text{assert } \text{expr})
\]

\[
(\text{verify } \text{expr})
\]

\[
(\text{debug } [\text{type ...}+] \text{ expr})
\]

\[
(\text{solve } \text{expr})
\]

\[
(\text{synthesize}
 \text{ #:forall } \text{expr}
 \text{ #:guarantee } \text{expr})
\]

\[> (\text{define (new-x)}
   \text{(define-symbolic* } x \text{ integer?)})
   x)
\]

\[> (\text{new-x})
 x$0
\]

\[> (\text{new-x})
 x$1
\]

\[> (\text{eq? (new-x) (new-x)})
 (= x$2 x$3)
\]

\[id \text{ is bound to a different constant every time define-symbolic* is evaluated.}
\]

**define-symbolic** creates a fresh symbolic constant of the given type and binds it to the variable id.

Symbolic values of a given type can be used just like concrete values of that type.
Rosette constructs: creating complex symbolic values

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

define-symbolic(*) can be used to create bounded symbolic instances of complex data types.
define-symbolic(*) can be used to create *bounded* symbolic instances of complex data types.

A concrete list of 4 symbolic integers; this is just a short-hand for evaluating define-symbolic* 4 times and collecting the results into a list.
Rosette constructs: creating complex symbolic values

`define-symbolic(*)` can be used to create **bounded** symbolic instances of complex data types.

\[
\begin{align*}
\text{define-symbolic} & (xs \text{ integer? } [4]) \\
\text{define-symbolic} & (xs) \\
\text{list} & (xs$0 \; xs$1 \; xs$2 \; xs$3) \\
\text{define-symbolic} & (len \text{ integer?}) \\
\text{take} & (xs \; len) \\
\{ & \begin{aligned}
[& (= 0 \; len$0) \; ()] \\
[& (= 1 \; len$0) \; (xs$0)] \\
[& (= 2 \; len$0) \; (xs$0 \; xs$1)] \\
[& (= 3 \; len$0) \; (xs$0 \; xs$1 \; xs$2)]
\end{aligned}
\end{align*}
\]

A symbolic list of length up to 4, consisting of symbolic integers.

- `(define-symbolic id type)`
- `(define-symbolic* id type)`
- `(assert expr)`
- `(verify expr)`
- `(debug [type ...+] expr)`
- `(solve expr)`
- `(synthesize #:forall expr #:guarantee expr)`
Rosette constructs: assert

assert checks that expr evaluates to a true value.

> (assert (>= 2 1)) ; passes
> (assert (< 2 1)) ; fails
assert: failed
**Rosette constructs: assert**

- `(define-symbolic id type)`
- `(define-symbolic* id type)`
- `(assert expr)`
- `(verify expr)`
- `(debug [type ...+] expr)`
- `(solve expr)`
- `(synthesize #:forall expr #:guarantee expr)`

**assert** checks that `expr` evaluates to a true value.

> `(assert (>= 2 1)) ; passes

> `(assert (< 2 1)) ; fails

**assert: failed**

> `(define-symbolic* x integer?)

> `(assert (>= x 1))

Symbolic `expr` gets added to the assertion store. Its meaning (true or false) is eventually determined by the solver in response to queries.
Rosette constructs: assert

(assert expr)
(asserts)
(define-symbolic id type)
(define-symbolic* id type)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)

assert checks that expr evaluates to a true value.

> (assert (>= 2 1)) ; passes
> (assert (< 2 1)) ; fails
assert: failed

> (define-symbolic* x integer?)
> (assert (>= x 1))
> (asserts)
(list (<= 1 x$0) ...)

Symbolic expr gets added to the assertion store. Its meaning (true or false) is eventually determined by the solver in response to queries.
Rosette constructs: from assert to verify

Do poly and fact produce the same output on all inputs?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))

; some tests ...
> (same poly fact 0) ; pass
> (same poly fact -1) ; pass
> (same poly fact -2) ; pass
```
Rosette constructs: verify

**verify** searches for a binding of symbolic constants to concrete values that causes at least one assertion in **expr** to fail.

```scheme
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
 #:forall expr
 #:guarantee expr)
```

Do poly and fact produce the same output on all inputs?

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))

; some tests ...
> (same poly fact 0) ; pass
> (same poly fact -1) ; pass
> (same poly fact -2) ; pass
```
**Rosette constructs: verify**

*verify* searches for a binding of symbolic constants to concrete values that causes at least one assertion in *expr* to fail.

\[
\text{(define-symbolic } \text{id} \text{ type)}
\]
\[
\text{(define-symbolic* } \text{id} \text{ type)}
\]

\[
\text{(assert } \text{expr)}
\]

\[
\text{(verify } \text{expr)}
\]

\[
\text{(debug [type ...+] } \text{expr)}
\]
\[
\text{(solve } \text{expr)}
\]
\[
\text{(synthesize}
\]
\[
\text{ #:forall } \text{expr}
\]
\[
\text{ #:guarantee } \text{expr}
\]

Do poly and fact produce the same output on all inputs?

\[
\text{(define } \text{poly } \text{x)}
\]
\[
\text{ (+ (* x x x x) (* 6 x x x))}
\]
\[
\text{ (* 11 x x) (* 6 x))}
\]

\[
\text{(define } \text{fact } \text{x)}
\]
\[
\text{ (* x (+ x 1) (+ x 2) (+ x 2))}
\]

\[
\text{(define } \text{same } \text{p f x)}
\]
\[
\text{ (assert (= (p x) (f x))}
\]

\[
> \text{(define-symbolic } \text{i integer?)}
\]
\[
> \text{(verify } \text{(same poly fact i))}
\]
**Rosette constructs: verify**

`verify` searches for a binding of symbolic constants to concrete values that causes at least one assertion in `expr` to fail.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Do poly and fact produce the same output on all inputs?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))
```

> (define-symbolic i integer?)
> (verify (same poly fact i))
(model [i -6])

No! The solver finds a concrete *counterexample* to the assertion in `same`.
**Rosette constructs: verify**

*verify* searches for a binding of symbolic constants to concrete values that causes at least one assertion in *expr* to fail.

```
(define-symbolic id type)
(define-symbolic* id type)
(verify expr)
```

Do poly and fact produce the same output on all inputs?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
     (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))
```

```
> (define-symbolic i integer?)
> (define cex
    (verify (same poly fact i)))
> (evaluate i cex)
-6
```

We can store bindings in variables and evaluate arbitrary expressions against them.
**Rosette constructs: verify**

`verify` searches for a binding of symbolic constants to concrete values that causes at least one assertion in `expr` to fail.

```lisp
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
```

The assertions encountered while evaluating `expr` are removed from the asserts store once a query (such as `verify`) completes.

Do poly and fact produce the same output on all inputs?

```lisp
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic i integer?)
> (define cex
    (verify (same poly fact i)))
> (asserts)
(list)```
Rosette constructs: from verify to debug

Why do poly and fact output different values on the input -6?

(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))
Rosette constructs: from verify to debug

**debug** searches for a minimal set of expressions of the given types that cause the evaluation of **expr** to fail.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
```

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
    (* 11 x x) (* 6 x x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))
```

Why do **poly** and **fact** output different values on the input -6?

```
(debug)
```
**Rosette constructs: debug**

**debug** searches for a minimal set of expressions of the given types that cause the evaluation of expr to fail.

- `(define-symbolic id type)`
- `(define-symbolic* id type)`
- `(assert expr)`
- `(verify expr)`
- `(debug [type ...+] expr)`
- `(solve expr)`
- `(synthesize #:forall expr #:guarantee expr)`

**To use debug**, require the debugging libraries, mark fact as the candidate for debugging, save the module to a file, and issue a debug query.

```lisp
(require rosette/query/debug rosette/lib/render)
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
(define/debug (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))
(define (same p f x)
  (assert (= (p x) (f x))))

> (render ; visualize the result
  (debug [integer?]
    (same poly fact -6)))
```

Why do poly and fact output different values on the input -6?
**Rosette constructs: debug**

**debug** searches for a minimal set of expressions of the given types that cause the evaluation of expr to fail.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

To use **debug**, require the debugging libraries, mark fact as the candidate for debugging, save the module to a file, and issue a **debug** query.

```
> (render ; visualize the result
  (debug [integer?] (same poly fact -6)))
```

Why do poly and fact output different values on the input -6?

```
(define/poly (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define/debug (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))
```

```
(define (same p f x)
  (assert (= (p x) (f x))))
```
Rosette constructs: from debug to solve

Can we repair fact on the input -6 as suggested by debug?

(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (* x (+ x 1) (+ x 2) (+ x 2)))

(define (same p f x)
  (assert (= (p x) (f x))))
Rosette constructs: from debug to solve

**solve** searches for a binding of symbolic constants to concrete values that causes all assertions in **expr** to pass.

**(define-symbolic id type)**
**(define-symbolic* id type)**
**(assert expr)**
**(verify expr)**
**(debug [type ...+] expr)**
**(solve expr)**
**(synthesize #:forall expr #:guarantee expr)**

Can we repair **fact** on the input -6 as suggested by **debug**?

**(define (poly x))**
\[+ (\ast x x x x) (\ast 6 x x x) (\ast 11 x x) (\ast 6 x)]

**(define (fact x))**
\[\ast x (+ x 1) (+ x 2) (+ x 2)]

**(define (same p f x))**
**(assert (= (p x) (f x)))**
**Rosette constructs: solve**

`solve` searches for a binding of symbolic constants to concrete values that causes all assertions in `expr` to pass.

```scheme
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Can we repair `fact` on the input -6 as suggested by `debug`?

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic* c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (solve (same poly fact -6))
```
Rosette constructs: solve

**solve** searches for a binding of symbolic constants to concrete values that causes all assertions in **expr** to pass.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Can we repair fact on the input -6 as suggested by **debug**?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic* c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (solve (same poly fact -6))

(model [c1$0 -66] [c2$0 7] [c3$0 7])
```

Yes! The solver finds concrete values for c1, c2, and c3 that work for the input -6.
Rosette constructs: solve many with define-symbolic*

**solve** searches for a binding of symbolic constants to concrete values that causes all assertions in **expr** to pass.

```lisp
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Can we repair fact on multiple inputs individually?

```lisp
(define (poly x)
 (+ (* x x x x) (* 6 x x x)
  (* 11 x x) (* 6 x)))

(define (fact x)
 (define-symbolic* c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (solve (begin
          (same poly fact -6)
          (same poly fact 12)))

(model [c1$1 -66] [c2$1 7] [c3$1 7]
       [c1$2 2508] [c2$2 -11] [c3$2 -11])
```

Solving same for multiple inputs: note the behavior of define-symbolic*.
Rosette constructs: solve many with `define-symbolic`

`solve` searches for a binding of symbolic constants to concrete values that causes all assertions in `expr` to pass.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
 #:forall expr
 #:guarantee expr)
```

Can we repair `fact` on multiple inputs simultaneously?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))
```

```
> (solve (begin
           (same poly fact -6)
           (same poly fact 12)))
  (model [c1 2] [c2 3] [c3 0])
```

Solving same for multiple inputs: note the behavior of `define-symbolic`. 
Can we repair `fact` on all inputs as suggested by `solve`?

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic c1 c2 c3 integer?)
  (+ x c1 (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))
```
Rosette constructs: synthesize

synthesize searches for a binding that causes all assertions in #:guarantee expr to pass for all bindings of the symbolic constants in the #:forall expr.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

Can we repair fact on all inputs as suggested by solve?

```
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))

> (define-symbolic* i integer?)
> (synthesize #:forall i #:guarantee (same poly fact i))
```
Rosette constructs: synthesize

`synthesize` searches for a binding that causes all assertions in `#:guarantee expr` to pass for all bindings of the symbolic constants in the `#:forall expr`.

```scheme
(define (poly x)
  (+ (* x x x x) (* 6 x x x)
      (* 11 x x) (* 6 x)))

(define (fact x)
  (define-symbolic c1 c2 c3 integer?)
  (* (+ x c1) (+ x 1) (+ x c2) (+ x c3)))

(define (same p f x)
  (assert (= (p x) (f x))))
```

Can we repair fact on all inputs as suggested by `solve`?

```scheme
> (define-symbolic* i integer?)
> (synthesize
  #:forall expr
  #:guarantee expr)
```

Yes! The solver finds concrete values for `c1`, `c2`, and `c3` that work for every input `i`.

```scheme
(model [c1 3] [c2 0] [c3 2])
```
**Rosette constructs: synthesize**

`synthesize` searches for a binding that causes all assertions in `#:guarantee expr` to pass for all bindings of the symbolic constants in the `#:forall expr`.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

To generate code, require the sketching library, save the module to a file, and issue a `synthesize` query.

```
> (define-symbolic* i integer?)
> (print-forms ; print the generated code
  (synthesize
    #:forall i
    #:guarantee (same poly fact i)))
```
Rosette constructs: synthesize

**synthesize** searches for a binding that causes all assertions in #:guarantee expr to pass for all bindings of the symbolic constants in the #:forall expr.

```
(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(verify expr)
(debug [type ...+] expr)
(solve expr)
(synthesize
 #:forall expr
 #:guarantee expr)
```

Can we repair fact on all inputs as suggested by solve?

```
(require rosette/lib/synthax)
(define (poly x)
 (+ (* x x x x) (* 6 x x x)
 (* 11 x) (* 6 x)))

(define (fact x)
 (* (+ x 3) (+ x 1) (+ x 0) (+ x 2)))

(define (same p f x)
 (assert (= (p x) (f x))))

> (define-symbolic* i integer?)
> (print-forms ; print the generated code
  (synthesize
   #:forall i
   #:guarantee (same poly fact i)))
```
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**

(1) **getting started** and (2) going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Common pitfalls and gotchas

“A gotcha is a valid construct in a system, program or programming language that works as documented but is counter-intuitive and almost invites mistakes because it is both easy to invoke and unexpected or unreasonable in its outcome.”

—Wikipedia

Reasoning precision
Unbounded loops
Unsafe features
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer \( k > 0 \) or \#f for approximate or precise reasoning, respectively.
Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.

• Controlled by setting current-bitwidth to an integer \( k > 0 \) or \(#f\) for approximate or precise reasoning, respectively.

; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer k > 0 or #f for approximate or precise reasoning, respectively.

```scheme
; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
```
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer $k > 0$ or #f for approximate or precise reasoning, respectively.

```scheme
; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
```
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting \texttt{current-bitwidth} to an integer \( k > 0 \) or \#f for approximate or precise reasoning, respectively.

; default current-bitwidth is \#f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
(model [x 64])
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer \( k > 0 \) or \#f for approximate or precise reasoning, respectively.

```scheme
; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
(model [x 64])
> (current-bitwidth 5)
> (solve (assert (= x 64)))
```
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.

• Controlled by setting current-bitwidth to an integer $k > 0$ or #f for approximate or precise reasoning, respectively.

```scheme
; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
(model [x 64])
> (current-bitwidth 5)
> (solve (assert (= x 64)))
(model [x 0])
> (verify (assert (not (= x 64))))
(model [x 0])
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

• Loops and recursion must be bounded (aka self-finitizing) by
  • concrete termination conditions, or
  • upper bounds on size of iterated (symbolic) data structures.
• Unbounded loops and recursion run forever.
Loops and recursion must be bounded (aka self-finitizing) by
• concrete termination conditions, or
• upper bounds on size of iterated (symbolic) data structures.

Unbounded loops and recursion run forever.

(define (search x xs)
  (cond
   [(null? xs) #f]
   [(equal? x (car xs)) #t]
   [else (search x (cdr xs))])))

> (define-symbolic xs integer? [5])
> (define-symbolic xl i integer?)
> (define ys (take xs xl))
> (verify
   (when (<= 0 i (- xl 1))
     (assert (search (list-ref ys i) ys)))))
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be *bounded* (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

```
(define (search x xs)
  (cond
    [(null? xs) #f]
    [(equal? x (car xs)) #t]
    [else (search x (cdr xs))]))
```

```scheme
> (define-symbolic xs integer? [5])
> (define-symbolic xl i integer?)
> (define ys (take xs xl))
> (verify
   (when (<= 0 i (- xl 1))
     (assert (search (list-ref ys i) ys))))
(unsat)
```

Terminates because search iterates over a bounded structure.
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

(define (factorial n)
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1)))]))
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

• Loops and recursion must be **bounded** (aka self-finitizing) by
  • concrete termination conditions, or
  • upper bounds on size of iterated (symbolic) data structures.
• Unbounded loops and recursion run forever.

```
(define (factorial n)
  (cond
   [(= n 0) 1]
   [else (* n (factorial (- n 1)))])
)
```

> (define-symbolic k integer?)
> (solve
>   (assert (> (factorial k) 10)))
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

```
(define (factorial n)
  (cond
   [(= n 0) 1]
   [else (* n (factorial (- n 1)))]))
```

```
> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k) 10)))
```

Unbounded because factorial termination depends on k.
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be **bounded** (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

```lisp
(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))])
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

```
(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))]))

> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k 3) 10)))
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

```scheme
(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))]))
```

> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k 3) 10)))

(unsat)

UNSAT because the bound is too small to find a solution.
Common pitfalls and gotchas: unbounded loops

Reasoning precision
Unbounded loops
Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

```scheme
(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))])
)
```

> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k 4) 10)))

(model
  [k 4])

Make sure the bound is large enough …
Common pitfalls and gotchas: unsafe features

Reasoning precision
Unbounded loops
Unsafe features

- Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`.
- Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

- Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`
- Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

```scheme
; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
```
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

• Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in #lang rosette/safe

• Unlifted constructs can be used in #lang rosette but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

; vectors are lifted
> `(define v (vector 1 2))
> `(define-symbolic k integer?)
> `(vector-ref v k)
`(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2)))
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

• Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in #lang rosette/safe

• Unlifted constructs can be used in #lang rosette but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2))

; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
Common pitfalls and gotchas: unsafe features

- Reasoning precision
- Unbounded loops
- Unsafe features
  - Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`
  - Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

```scheme
; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (∼ (= 0 k) 1) (∼ (= 1 k) 2))

; hashes are unlifted
> (define h (make-hash '(((0 . 1)(1 . 2)))))
> (hash-ref h k)
hash-ref: no value found for key
key: k
```
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

• Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in #lang rosette/safe

• Unlifted constructs can be used in #lang rosette but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (⊢ (= 0 k) 1) (⊢ (= 1 k) 2)))

; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
hash-ref: no value found for key
  key: k
> (hash-set! h k 3)
> (hash-ref h k)
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

- Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in \texttt{#lang rosette/safe}

- Unlifted constructs can be used in \texttt{#lang rosette} but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

\begin{verbatim}
; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2)))

; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
hash-ref: no value found for key
key: k
> (hash-set! h k 3)
> (hash-ref h k)
3
\end{verbatim}
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

Solver-aided programming in two parts: (1) **getting started** and (2) going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.

emina.github.io/rosette/