Rosette Tutorial

By
Abdul Rafae Noor

What is Rosette

- Rosette is a solver-aided programming language with constructs for
  - Program Synthesis
  - Program Verification (with user-defined equivalence constraints)
  - Optimization
  - Angelic Execution

- Comes pre-packaged with multiple solvers such operating on various domains
  - E.g. Theory Of BitVectors, Integers

- Extends the Racket Functional programming language
  - LISP, Scheme like syntax

The Rosette Language

BitVector Basics

```
; Naming a typedef for specific bitvector length type
(define int128? (bitvector 128))
(define int32? (bitvector 32))
(define int8? (bitvector 8))
(define int4? (bitvector 4))

(define (int4 i)
  (bv i int4?))

(define (int8 i)
  (bv i int8?))

(define (int32 i)
  (bv i int32?))

(define (int128 i)
  (bv i int128?))


(println (int128 10))
;; Output: (bv #x0000000000000000000000000000000a 128)
```

BitVector Basics

```
;; Rosette provides various bitvector operations on bitvectors

(println (bvadd (int32 1) (int32 3)))
;; Output: (bv #x00000004 32)

(println (bvmul (int32 2) (int32 3)))
;; Output: (bv #x00000006 32)
```

BitVector Basics

```
;; Most bitvector operations are
;; have take either two operands or
;; one.  We can fold the binary operations
;; on lists of arbitrary lengths


(define bv_list (list (int32 0) (int32 1) (int32 2) (int32 3) (int32 4)))


;; Apply folds an operation
;; on a list of values
(define (simple-add-reduce ls)
  (apply bvadd ls)
  )

(println (simple-add-reduce bv_list))
;; Output: (bv #x0000000a 32)

(println (bitvector->integer (simple-add-reduce bv_list)))
;; Output: 10
```

BitVector Basics

```
(define concat_vec
  (concat (int32 4) (int128 3)))


(println concat_vec)
;; Output:  (bv #x0000000400000000000000000000000000000003 160)
```

```
(define value
  (bv #x11112222 (bitvector 32)))

(println value)
;; Output: (bv #x11112222 32)


;; Extracting the bitvector starting from
;; 8 bits from the right up till 23 bits
;; from the right yielding a
;; bitvector of length (23-8) + 1 = 16 bits
(define slice (extract 23 8 value))

(println slice)
;; Output: (bv #x1122 16)
```

- *Many more operations such as SEXT, ZEXT, OR, AND, etc.*

Symbolic Exec Basics

```
;; Rosette provides a new define-symbolic keyword
;; for defining symbolic values. These values can
;; be integers or bitvectors (with differennt underlying
;; theories).
(define-symbolic _x (bitvector 32))

(println _x)
;; Output:  _x
```

Symbolic Exec Basics

```
;; Rosette hides away the complexity of
;; using symbolic/concrete variant of bitvector
;; operations by exposing the same name for
;; bitvector operations for both.

(define y
  (bv 1 (bitvector 32)))

(println (bvadd _x y))
;; Output: (bvadd (bv #x00000001 32) _x)
```

Symbolic Exec Basics

```
;; Let's say we want to define a query to return
;; to lowest common multiple of two bitvectors, but using
;; falling back on symbolic evaluation to solve it.

(define-symbolic _lcm (bitvector 32))

(define (symbolic-lcm-first a b)
  (define zero (bv 0 (bitvector 32)))
  (assume (bvsgt _lcm zero))

  (solve (begin
           (assert (equal? zero (bvsrem _lcm a)))
           (assert (equal? zero (bvsrem _lcm b)))
                   ))
          )
```

Symbolic Exec Basics

```
;; Let's say we want to define a query to return
;; to lowest common multiple of two bitvectors, but using
;; falling back on symbolic evaluation to solve it.

(define-symbolic _lcm (bitvector 32))

(define (symbolic-lcm-first a b)
  (define zero (bv 0 (bitvector 32)))
  (assume (bvsgt _lcm zero))

  (solve (begin
           (assert (equal? zero (bvsrem _lcm a))
           (assert (equal? zero (bvsrem _lcm b))
                   ))
         )
```

Symbolic Exec Basics

```
(define res
  (symbolic-lcm-first (bv 5 (bitvector 32)) (bv 7 (bitvector 32)))
  )

(println res)
;; Output: (model
;;  [_lcm (bv #x302f6941 32)])

(assert (sat? res) "Unsatisfiable query")
```

Symbolic Exec Basics

```
;; assign lcm_val the concrete value
;; for _lcm according to the 'model'
;; generated.
(define lcm_val (evaluate _lcm res))

(println lcm_val)
;; Output: (bv #x302f6941 32)

;; Converting to integer
(println (bitvector->integer lcm_val))

;; Output: 808413505
```

Symbolic Exec Basics

- 808413505 is not the LCM for 7 and 5!
  - *What went wrong?*

Symbolic Exec Basics

- 808413505 is not the LCM for 7 and 5!
  - *What went wrong?*

```
;; Let's say we want to define a query to return
;; to lowest common multiple of two bitvectors, but using
;; falling back on symbolic evaluation to solve it.

(define-symbolic _lcm (bitvector 32))

(define (symbolic-lcm-first a b)
  (define zero (bv 0 (bitvector 32)))
  (assume (bvsgt _lcm zero))

  (solve (begin
           (assert (equal? zero (bvsrem _lcm a)))
           (assert (equal? zero (bvsrem _lcm b)))
                   ))
          )
```

Symbolic Exec Basics

```
;; The constraints were under specified. There
;; are multiple values which the remainder returns
;; zero. We want the smallest possible value in
;; that set.
```

```
;; We use another symbolic construct
;; to include the constraint that we
;; want to minimize the specific
;; multiple value.
```

Symbolic Exec Basics

```
(define (symbolic-lcm a b)
  (define zero (bv 0 (bitvector 32)))
  (assume (bvsgt _lcm2 zero)) ;; Assume Positive
  (define result
    (optimize
      #:minimize (list  (bvsub _lcm2 a) (bvsub  _lcm2 b))
      #:guarantee  (begin
              (assert (equal? zero (bvsrem _lcm2 a)))
              (assert (equal? zero (bvsrem _lcm2  b)))
              )
    )
  )
  result
```

Minimize LCM difference from input values

=>

Smallest Multiple Value

Symbolic Exec Basics

```
(println lcm_7_5)

;; Output: (model
;; [_lcm2 (bv #x00000023 32)])


(assert (sat? res) "Unsatisfiable query")

(define final-res (evaluate _lcm2 lcm_7_5))

(println (bitvector->integer final-res))
;; Output: 35!
```

Success!

Programming By Example

- We have a list of input data and output data

- Can we synthesize a polynomial expression on input variables to calculate output?

Row ID	a	b	c	output
1	1	1	1	5
2	2	5	6	33
3	0	0	0	0

Programming By Example

```
(define a_1 1)    (define a_2 2)     (define a_3 0)
(define b_1 2)    (define b_2 5)     (define b_3 0)
(define c_1 1)    (define c_2 6)     (define c_3 0)
(define res1 5)   (define res2 33)   (define res3 0)
```

Defining names for data points

Programming By Example

```
(define-grammar (poly-grammar a b c)

                  [expr
                    ( choose
                      a
                      b
                      c
                      (+ (expr) (expr))
                      (- (expr) (expr))
                      (/ (expr) (expr))
                      (* (expr) (expr))
                      )]

                  )
```

Programming By Example

- Grammar Tree can have infinite depth!
  - Synthesis may become infeasible due to infinite recursion

- **Solution**: Limit the recursive definition to a bounded depth

```
(define (synth_grammar arg1 arg2 arg3)
                        (poly-grammar arg1 arg2 arg3 #:depth 3))
```

Programming By Example

```
(define sol
    (synthesize
    #:forall (list a_1 b_1 c_1 res1 a_2 b_2 c_2 res2 a_3 b_3 c_3 res3 )
    #:guarantee  (begin
            (assert
                (equal? res1 (synth_grammar a_1 b_1 c_1) ))
            (assert
                (equal? res2 (synth_grammar a_2 b_2 c_2) ))

            (assert
                (equal? res3 (synth_grammar a_3 b_3 c_3) ))
```

Programming By Example

```
;; Output:
;; '(define (synth_grammar arg1 arg2 arg3)
;;    (+ (+ (+ arg2 arg2) (* arg3 arg3)) (- (- arg2 arg3) (* arg1 arg3))))
```

Success!

BitVector Midpoint

- Given two BitVectors *lo* and *hi*, find the midpoint between them

```
; Returns the midpoint of the interval [lo, hi].
(define (bvmid lo hi)  ; (lo + hi) / 2
  (bvsdiv (bvadd lo hi) (int32 2)))
```

BitVector Midpoint

```
(define (check-mid impl lo hi)      ; Assuming that
  (assume (bvsle (int32 0) lo))     ; 0 ≤ lo and
  (assume (bvsle lo hi))            ; lo ≤ hi,
  (define mi (impl lo hi))          ; and letting mi = impl(lo, hi) and
  (define diff                      ; diff = (hi - mi) - (mi - lo),
    (bvsub (bvsub hi mi)
           (bvsub mi lo)))          ; we require that
  (assert (bvsle lo mi))            ; lo ≤ mi,
  (assert (bvsle mi hi))            ; mi ≤ hi,
  (assert (bvsle (int32 0) diff))   ; 0 ≤ diff, and
  (assert (bvsle diff (int32 1))))  ; diff ≤ 1.
```

BitVector Midpoint

```
(define-symbolic low high int32?)

(define cex (verify (check-mid bvmid low high)))
(println cex)
```

BitVector Midpoint

```
(define-symbolic low high int32?)

(define cex (verify (check-mid bvmid low high)))
(println cex)
;; Output: (model
;;  [low (bv #x394f0402 32)]
;;  [high (bv #x529e7c00 32)])
```

Concrete Counter Example which failed verification!

BitVector MidPoint

```
(define-grammar (fast-int32 x y)  ; Grammar of int32 expressions over two inputs:
  [expr
   (choose x y (?? int32?)        ; <expr> := x | y | <32-bit integer constant> |
           ((bop) (expr) (expr))  ;            (<bop> <expr> <expr>) |
           ((uop) (expr)))]       ;            (<uop> <expr>)
  [bop
   (choose bvadd bvsub bvand      ; <bop>  := bvadd  | bvsub | bvand |
           bvor bvxor bvshl       ;            bvor   | bvxor | bvshl |
           bvlshr bvashr)]        ;            bvlshr | bvashr
  [uop
   (choose bvneg bvnot)])         ; <uop>  := bvneg | bvnot
```

```
'define
'(bvmid-fast lo hi)
(list 'bvlshr '(bvadd hi lo) (bv #x00000001 32)))
```

Conclusion

- Abstracts away SMT formula generation for problem description
  - E.g. description can contain loops and other racket constructs

- Equivalence description entirely expressed in Rosette.
  - Can provide formal verification
  - Can provide reference implementation and verify translation is semantically equivalent.

- Even better, tell Rosette to synthesize equivalent translation in your own DSL!
  - (DSL defined in Rosette)

- CEGIS Algorithm at the core of synthesis
  - Learns from mistakes and avoids making the same mistake again

Resources

Rosette Website:

https://docs.racket-lang.org/rosette-guide/index.html

Tutorial Code from slide:

https://github.com/RafaeNoor/Rosette-Tutorial

Program Synthesis Introduction:

https://people.csail.mit.edu/asolar/SynthesisCourse/TOC.htm

https://people.csail.mit.edu/asolar/SynthesisCourse/Lecture10.htm (CEGIS)