Session 8: Z3 Tutorial and More on Racket

Z3 Tutorial

> references:
>
> - Z3 Online Tutorial

- What does a solver do?

It can be used to check the satisfiability of logical formulas over one or more theories. (Or you can think of it as a tool for solving equations, with some more helpful features.)

- So, let solve x in $x^2 = 1$ using Z3:

```
(declare-const x Int)
(assert (= (* x x) 1))
(check-sat)
(get-model)
```

How about another solution for x?

```
(declare-const x Int)
(assert (= (* x x) 1))
(assert (not (= x 1)))
(check-sat)
(get-model)
```

Is there any other solution?

```
(declare-const x Int)
(assert (= (* x x) 1))
(assert (not (= x 1)))
(assert (not (= x -1)))
(check-sat)
(get-model)
```

- working with bitvectors in Z3 solver

```
(assert (= #x07 #x03))
(check-sat)
```

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert (= (bvor x y) (bvnot (bvand x y))))
(check-sat)
```

- find more examples in the tutorial

Racket: Pattern Matching `guide.rkt`

> references:
>
> - Pattern Matching
> - Quasiquoting
> - For
> - What is the difference between quote and list?

- Example: reverse a list of 3 elements

```
(define (rev3 l)
  (match l
    [(list a b c) (list c b a)]
    [_ (error "list is not of length 3")]))
```

- Example: reverse an arbitary list

```
(define (rev l)
  (match l
    [(list ) (list )]
    [(list head tail ...) (append (rev tail) (list head))]
    [_ (error "not a list")]))
```

- Example: find maximum integer from a list

```
(define (find-max l)
  (match l
    [(list head tail ...)
     (let ([r (find-max tail)])
       (if (integer? r) (max head r) head))]
    [_ #f]))
```

- Example: correctly representing a list

```
> ; are these the same?
> '(1 2 3)
> `(1 2 3)
> (list 1 2 3)
```

```
> ; are these the same?
> '(1 1+1 1+1+1)
> '(1 (+ 1 1) (+ 1 1 1))
> `(1 (+ 1 1) (+ 1 1 1))
> (list 1 (+ 1 1) (+ 1 1 1))
> (eval `(list 1 (+ 1 1) (+ 1 1 1)))
```

```
> ; are these the same?
> `(1 ,(+ 1 1) 3)
> '(1 2 3)
> (list 1 2 3)
```

- Example: matching symbols, e.g., define a fancy `printf` function that works like:

```
> (Mike "Hello")
Mike says Hello.
> (Alice "Bye")
Alice says Bye.
```

Notice that ideally we should wrap our fancy function into syntax definition like what `bv.rkt` is doing, but here as a demonstration we just wrap our commands in quasiquotes.

```
(define (fancy-says x)
  (match x
    [`(,who ,what) (printf (string-append (symbol->string who) " says " what
".\n"))]
    [_ (error "syntax error")]))
```

So in the demo we should call it like this:

```
> (fancy-says `(Mike "Hello"))
Mike says Hello.
> (fancy-says `(Alice "Bye"))
Mike says Bye.
```

- Example: matching symbols, e.g., create your own syntax rule `(my-list elem ...+)` for `list` using pattern matching

```
; assume that there's no nested list and the list only contains integers (no need
for eval)
(define (read-my-list-0 x)
  (match x
    [`(my-list ,a ...) a]
    [_ (error "syntax error")]))
```

```
; assume that there are nested lists with different types (need to call eval)
(define (read-my-list-1 x)
  (match x
    [`(my-list ,a ...) (for/list ([i a]) (read-my-list-1 i))]
    [b (eval b)]))
```

Other Notes for HW5

- The Racket Documentation is very useful. Search it whenever you have any questions!
- access an element in a list/string (even though it's not recommended by Racket)

```
> (list-ref (list 1 2 3 4) 0)
1
> (string-ref "Hello World" 1)
#\e
```

- type conversion

```
> (integer->char 65)
#\A
> (string->list "ABCDE")
'(#\A #\B #\C #\D #\E)
```

- indexing

```
> (index-of (list 9 8 7 6) 7)
2
```

- "Racket style" for loop

```
> (for/list ([i (list 1 2 3)]) (+ i 1))
'(2 3 4)
```

- getting the sample smt formulas (see `smt.rkt`)

```
#lang rosette

;this will get the smt formulas dumped when you call the verify function in
Rosette
(output-smt #t)

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

(define (factored 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 factored i )))
```

Useful Resources

Beautiful Racket

Racket Summary

Racket Documentation