A Taste of Racket and PLT Redex

Table of Contents

Scheme Basics

Function Abstraction

Scheme Basics

Function Abstraction

```
(λ (x y)
   (+ x y))
```

Scheme Basics

Function Abstraction

```
(λ (x y)
   (+ x y))
```

Function Application

Scheme Basics

Function Abstraction

```
(λ (x y)
  (+ x y))
```

Function Application

```
((λ (x)
   (+ x 1)) 2)
```

Scheme Basics

Function Abstraction

```
(λ (x y)
   (+ x y))
```

Function Application

```
((λ (x)
    (+ x 1)) 2)
```

Naming

Scheme Basics

Function Abstraction

```
(λ (x y)
  (+ x y))
```

Function Application

```
((λ (x)
   (+ x 1)) 2)
```

Naming

```
(define foo 1)
(define add1
  (λ (x)
    (+ x 1)))
```

Macro System

```
(define-syntax (time-it stx)
  (syntax-parse stx
    [(_ task)
     #'(thunk-time-it (λ () task))]))
(define (thunk-time-it task)
  (define before (cim))
  (define answer (task))
  (define delta  (- (cim) before))
  (printf "time: ~a ms\n" delta)
  answer)
(define cim current-inexact-milliseconds)
```

Macro System

```
(define-syntax (time-it stx)
  (syntax-parse stx
    [(_ task)
     #'(thunk-time-it (λ () task))]))
(define (thunk-time-it task)
  (define before (cim))
  (define answer (task))
  (define delta  (- (cim) before))
  (printf "time: ~a ms\n" delta)
  answer)
(define cim current-inexact-milliseconds)
```

Run it

```
(time-it (add1 2))
```

What is PLT Redex

- A tool to explore and experiment with languages
- It can formalize a language and more powerful than prolog
- It can test whether your judgment holds or not
- It can draw the derivation tree for your examples
- It can generate random terms according to your specified constraints
- Thus it can test the property of your language instead of proving it

STLC Syntax

```
(define-language L
  (x ::= variable-not-otherwise-mentioned)
  (e ::= x (λ (x : τ) e) (e e)
     false true (if e then e else e))
  (τ ::= bool (τ -> τ))
  (Γ ::= ((x τ) ...))
  (v ::= true false (λ (x : τ) e))
  (E ::= hole (E e) (v E) (if E then e else e))
  #:binding-forms
  (λ (x : τ) e #:refers-to x))
```

STLC Typing Judgment

```
(define-judgment-form L
  #:mode (typeof I I I O)
  #:contract (typeof Γ e : τ)
  [(lookup Γ x τ)
   ------------- "t-var"
   (typeof Γ x : τ)]
  [(typeof (ext Γ (x₁ τ₁)) e : τ)
   --------------------------- "t-abs"
   (typeof Γ (λ (x₁ τ₁) e) : (τ₁ -> τ))]
  [(typeof Γ e₁ : (τ₁ -> τ₂))
   (typeof Γ e₂ : τ₁)
   ------------------------ "t-app"
   (typeof Γ (e₁ e₂) : τ₂)]
  [------------------------ "t-true"
   (typeof Γ true : bool)]
  [------------------------ "t-false"
   (typeof Γ false : bool)]
  [(typeof Γ e₁ : bool)
   (typeof Γ e₂ : τ)
   (typeof Γ e₃ : τ)
   ---------------------- "t-if"
   (typeof Γ (if e₁ then e₂ else e₃) : τ)])
```

Test whether a judgment holds or not

```
(test-equal (judgment-holds
             (typeof () true : τ) τ)
            (list (term bool)))
(test-equal (judgment-holds
             (typeof ((y bool)) y : τ) τ)
            (list (term bool)))
```

Draw a derivation tree

```
(show-derivations
 (build-derivations
  (typeof ((y bool)) y : bool)))
```

Generate terms

```
(redex-check
 L v
 (redex-match? L e (term v))
 #:attempts 1000)
```