Descrição da linguagem "plai-typed"

Marco Dimas Gubitoso

28 de fevereiro de 2014

Sumário

A linguagem *plai-typed* é uma variante de *racket* com verificação de tipos e construtores especiais, além de procedimentos de testes embutidos. Há uma redução do número pré-definido de procedimentos e um aumento no rigor das expressões, já que esta linguagem é fortemente tipada.

Com isso, a sintaxe se torna mais rígida e algumas das expressões normais em *scheme*, *lisp* e *racket* são substituídas ou ganham uma semântica ligeiramente diferente. Este documento destaca estas mudanças.

Em *plai-typed*, o programa é composto por uma série de definições e expressões. As expressões são compostas de listas e dados atômicos.

1 Dados atômicos

São correspondentes aos valores nativos de uma linguagem, a exceção aqui são aos procedimentos, que veremos mais tarde.

Esta é a lista dos átomos possíveis.

Booleanos `true`, `false`, `#t`, `#f`

Números `1`, `42`, `4.23`, `2/3`, `4+5i`

Strings `"uma coisa"`, `"outra"`

Símbolos `'algo`, `'$eitcha!!:`

Caracteres `#\a`, `#\space`, `#\u2232`

1.1 Funções prontas para dados atômicos

As usuais, mas com controle de tipo. Devem aparecer na primeira posição da lista. Estes são alguns exemplos:

- `not`, `and`, `or`
- `+`, `-`, `*`, `/`
- `<`, `>`, `=`, etc (numéricos)
- `string=?`, `char=?`
- `equal?` — mesmo resultado
- `eq?` — mesma estrutura e resultado
- `string-append`, `string-ref`

1.2 S-expressões

Em *plai*, tudo o que corresponde a uma entrada para interpretação é uma *s-expression* (um tipo especial). Isso vale para qualquer expressão "*quoted*", isto é, com "'" na frente, como `'(+ 1 2)`, com exceção dos símbolos.

É possível converter tipos:

- `(symbol->s-exp 'hoje)`

- `(number->s-exp 23)`
- `(s-exp->number '23)`

1.3 Dados compostos prontos

Vamos ver os detalhes ao longo da disciplina, mas alguns já podem ser entendidos.

1.3.1 Listas

- `empty`, note que `'()` é uma *s-expression*
- `(list 2 4 4)`, é do tipo *(listof number)*
- `(list "sim" "não"`, *(listof string)*
- `(cons 0 (list 1 2)`, gera uma `(list 0 1 2)`, *(listof number)*
- `(list (list 2 3) (list 4))`, *(listof (listof number))*
- `(append (list 2 3) (list 3 3))`, gera `(list 2 3 3 3)`
- `first` é o `car` tipado, não existe mais o `car` original
- `rest` é o `cdr`
- De modo similar, exite o `second`, `third` e `fourth`
- `list-ref` recebe um parâmetro adicional para retornar um elemento qualquer da lista

1.3.2 Vetores

Vetores (*vector*s) são listas de tamanho fixo, com acesso eficiente. Também são tipados.

- `(vector 1 2 3 4)`, tipo *(vectorof number)*
- `(vector-ref (vector 42 2) 0)`, retorna 42

Existe um caso particular, `box`, que será visto mais tarde.

1.3.3 Valores

values são tuplas genéricas. Podem conter qualquer combinação de valores, como uma `struct` da *C*.

- `(values 90 3/4)`, tipo *(vector-immutable)*
- `(values (vector 42 2) 0)`, idem

2 Definições

A maior diferença nas definições "comuns" é a possibilidade de especificar um tipo:

```
(define x : number 3)
(define y 4)
```

Nestes exemplos, `x` tem o tipo *number* pré-declarado. O tipo de `y` foi inferido. **Não** é possível misturar tipos.

Com *values* é possível fazer definições em paralelo.

```
(define-values (marca preço) (values "Taipa" 43.23))
```

2.1 Procedimentos ou funções

Para definir proceimentos com `define`, basta colocar os argumentos, como em *racket*, mas agora tanto os argumentos, como a função podem ter tipos. O tipo de cada argumento, quando declarado, deve ser fornecido por uma dupla (variação de `cons`) separada por '`:`'.

Este exemplo define um procedimento que triplica o valor de seu argumento, seu tipo é *(number -¿ number)*.

```
(define (3* [ n : number]) : number (* 3 n))
```

As funções têm recursão e fechamento. Este é um exemplo interessante:

```
(define (is-odd? x)
  (if (zero? x)
      false
      (is-even? (- x 1))))
```

```
(define (is-even? x)
  (if (zero? x)
      true
      (is-odd? (- x 1))))
```

2.1.1 Funções de primeira classe

Funções de primeira classe são valores como outros quaisquer. Aparecem em todas as linguagens funcionais, normalmente construídas com `lambda` ou λ. A diferença em *plai-typed* é que há a restrição de tipos.

Para referência, este é um exemplo de uma função aplicada ao número 10:

```
((lambda (x) (+ x 1)) 10)
```

Definições com tipos:

```
(define mais2 : (number -> number) (lambda (x) (+ x 2)))
(define soma2 (lambda ([x : number]) : number (+ x  2)))
(define soma2 (lambda ([x : number]) : number (+ x  2)))
```

Outra característica de linguagens funcionais é o fechamento (*closure*). A função captura as definições existentes no momento de sua criação.

```
(define (cria-somador n)
  (lambda (m)
    (+ m n)))
(cria-somador 8)
(define soma-5 (cria-somador 5))
(soma-5 12)                  ; => 17
((cria-somador 5) 12)        ; => 17
```

Os procedimentos especiais `map` e `filter` também estão definidos, mas com a restrição de tipos. A implicação disso é que a lista deve ser declarada como tal.

```
(map (lambda (x) (* x x))  (list 1 2 3))
(filter (lambda (x) (> x 5) (list -2 0 321 1 4 90)))
```

2.2 Formas locais

Existem vários modos de limitar o escopo de associações. As associações com `define` são visíveis em todo o escopo.

O modo mais explícito é com `local`, que cria um novo escopo e espera uma lista de definições como primeiro argumento. Não é muito comum, exceto para algumas definições mais sofisticadas.

```
(local
  ([define v 20]
   [define l (lambda () 22)])
  ( + v (l))
)
```

As formas mais comuns são: `let`, `let*` e `letrec`.

`let` faz associações locais de símbolos a valores, veja nos exemplos abaixo. O `let*` faz com que as definições sejam encaixadas:

```
(let ([x 10]
      [y 11])
  (+ x y))                ; => 21

(let ([x 0])
  (let ([x 10]
        [y (+ x 1)])
    (+ x y)))             ; => 11

(let ([x 0])
  (let* ([x 10]
         [y (+ x 1)])
    (+ x y)))             ; => 21
```

O `letrec` permite que definições posteriores posssam ser usadas. Faz uma associação de cada identificador com um valor indefinido, que pode ser trocado no primeiro uso; pense em um protótipo de função:

```
(letrec ([is-even? (lambda (n)
                     (or (zero? n)
                         (is-odd? (sub1 n))))]
           [is-odd? (lambda (n)
```

```
                    (and (not (zero? n))
                         (is-even? (sub1 n))))])
    (is-odd? 11))
```

Existe ainda um tipo especial de `let`, chamado de *named let*. Serve para a construção de laços a partir da recursão de cauda.

```
(let loop ([x 10])
     (if (zero? x)
         (display "FIM\n")
         (begin
            (displayln x)
            (loop (- x 1)))))
```

Este código é equivalente a

```
(letrec ([L (lambda(x)
               (if (zero? x)
                   (display "FIM\n")
                   (begin
                        (displayln x)
                        (L (- x 1)))))
         ])
        (L 10))
```

Note que o `letrec` é necessário, pois `L` é usado dentro de sua própria definição.

2.3 Definição de tipos

Esta é talvez a parte mais interessante do *plai-typed*. Podemos definir tipos novos facilmente, tal como é feito em linguagens orientadas a objetos. A função que cria um novo tipo é `define-type`.

Cada tipo está associado a um conjunto de variantes, cujo nome é o próprio construtor. A declaração tem a seguinte forma:

```
(define-type TIPO
             [variante (campo : tipo)
                       (campo : tipo)
```

```
                    ⋮
                    (campo : tipo)]
          ⋮
)
```

Cada *variante* é o construtor, que recebe os argumentos declarados. A forma *variante-campo* devolve o valor correspondente ao campo selecionado, enquanto que *variante?* retorna se o argumento pertence ao tipo.

```
(define-type Shape
  [square (side : number)]
  [circle (radius : number)]
  [triangle (height : number)
            (width : number)])

(define (curvy? [s : Shape]) : boolean
  (circle? s))

(curvy? (square 5)) ; => #f
(curvy? (circle 5)) ; => #t
(curvy? (triangle 3 5)) ; => #f

(define torre (triangle 90 4))
(triangle-width torre)
```

A construção de tipos parametrizados (como *templates*) é possível com o uso de símbolos.

```
(define-type (Tree 'a)
  [leaf (val : 'a)]
  [node (left : (Tree 'a))
        (right : (Tree 'a))])
```

2.4 Comportamento definido por variante de tipo

O `type-case` define um comportamento para cada variante de um tipo, pense em polimorfismo.

```
(define-type Shape
  [square (side : number)]
  [circle (radius : number)]
  [triangle (height : number)
            (width : number)])

(define (área s)
  ( type-case Shape s
     [square (l) (* l l)]
     [circle (r) (* 3.141592654 (* r r))]
     [triangle (h w) (/ (* h w ) 2)]))
```