

Lecture 13: Algebraic and Union Types

David Bremner

February 28, 2024

define-type

p. 150

```
bt1 (define-type BT
      [mt]
      [node (v : Number) (l : BT) (r : BT)])
```

- ▶ Binds the name BT in the current type environment
- ▶ Supports recursion
- ▶ Defines two variants mt and node
- ▶ What are the types of mt and node?

Generated bindings

Constructors

```
(mt : ( -> BT))  
(node : (Number BT BT -> BT))
```

Predicates

```
(mt? : (BT -> Boolean))  
(node? : (BT -> Boolean))
```

Accessors

```
(node-v : (BT -> Number))  
(node-l : (BT -> BT))  
(node-r : (BT -> BT))
```

The problematic ones

p. 151

```
(node-v : (BT -> Number))  
(node-l : (BT -> BT))  
(node-r : (BT -> BT))
```

bt2

```
(define (size-wrong (t : BT))  
  (+ 1 (+ (size-wrong (node-l t))  
          (size-wrong (node-r t)))))  
(size-wrong (mt))
```

Safely accessing variants

p. 152

bt3

```
(define (size-pm t)
  (type-case BT t
    [(mt) 0]
    [(node v l r) (+ 1 (+ (size-pm l) (size-pm r)))]))
(size-pm (mt))
```

Typechecking type-case

Each define-type *extends* the type checker.

p. 153

$$\frac{\Gamma \vdash e : BT \quad \Gamma \vdash e_1 : T \quad \Gamma[V \leftarrow \text{Num}, L \leftarrow BT, R \leftarrow BT] \vdash e_2 : T}{\Gamma \vdash (\text{type-case } BT \ e \ [(mt) \ e_1] [(node \ V \ L \ R) \ e_2]) : T}$$

- ▶ can be automatically generated

Space usage for algebraic data type

p. 153

- ▶ Can desugar pattern matching into cond
- ▶ Need some level of “local” tags for predicates

bt4

```
(define (size-pm-ds (t : BT))
  (cond
    [(mt? t) 0]
    [(node? t)
     (let ([v (node-v t)]
           [l (node-l t)]
           [r (node-r t)])
       (+ 1 (+ (size-pm-ds l) (size-pm-ds r))))]))
```

Retrofitted type systems

- ▶ A common (recent) strategy is to add a static type system to existing dynamically typed languages
- ▶ Examples include `Typescript` and `mypy` for python.
- ▶ The general idea is convert some run-time errors into compile-time errors.

Example of retrofitted types: typed/racket

Type system accepts more, therefore sometimes needs more help.

```
trgood (define f
        (lambda ([x : Number] [y : Number])
          (if x
              (+ y 1) (+ y 2))))
```

```
trbad (lambda (x)
       (if x
           (+ x 1) (+ x 2)))
```

- ▶ still detects type-errors, but there are some subtle differences

typed/racket accepts more code

Without annotation, inferred types are weaker.

trnew

```
(define f
  (lambda (x y)
    (cond
      [(string? y) (string-append "hello " y)]
      [(number? y) (if x (+ y 1) (+ y 2))]
      [else (error 'f "unknown type")]))))
```

Another approach to variants

Variant information is lost to the type system.

p. 155

```
(node : (Number BT BT -> BT))
```

Union types

In “typed/racket” we have another option.

bt5

```
(define-type-alias BT (U mt node))  
(struct mt ())  
(struct node ([v : Number] [l : BT] [r : BT]))
```

- ▶ what constructors, predicates, and accessors are defined?

Using our union type

p. 156

bt6

```
(define t1
  (node 5
    (node 3
      (node 1 (mt) (mt))
      (mt))
    (node 7
      (mt)
      (node 9 (mt) (mt))))))
```

```
(node-l t1) ; OK
;;(node-l (node-l t1)) ; not OK
```

Computing the size

- ▶ Does the following typecheck? Why or why not?

p. 156

bt7

```
(define (size-tr [t : BT]) : Number
  (cond
    [(mt? t) 0]
    [(node? t) (+ 1
                  (size-tr (node-l t))
                  (size-tr (node-r t)))]))
```

Predicates are special in typed/racket

p. 157

- ▶ This program has a bug. Does the type checker catch it or not?

```
bt8 (define (size-tr [t : BT]) : Number
      (cond
        [(node? t) 0]
        [(mt? t) (+ 1
                    (size-tr (node-l t))
                    (size-tr (node-r t)))]))
```

Lecture 13: Algebraic and Union Types

└ Union types

└ Predicates are special in typed/racket

1. The type errors from typed/racket are much better than those from plait, at least here

- ▶ This program has a bug. Does the type checker catch it or not?

```
(define (size-tr [t : BT]) : Number
  (cond
    [(node? t) 0]
    [(mt? t) (+ 1
              (size-tr (node-l t))
              (size-tr (node-r t)))]))
```

If-Splitting

p. 157

```
(define (size-tr [t : BT]) : Number
  (cond
    [(mt? t) ...]
    [(node? t) ...]))
```

$$\frac{\Gamma \vdash e : (\bigcup T_1 \dots T_k) \quad \Gamma[e \leftarrow T_1] \vdash b_1 : T_1 \quad \dots \quad \Gamma[e \leftarrow T_k] \vdash b_k : T_k}{\Gamma \vdash (\text{cond} [(T_1? e) b_1] \dots [(T_k? e) b_k]) : (\bigcup T_1 \dots T_k)}$$

Option types and Unions 1/2

p. 158

```
div1 (define (safe-div num den)
      (if (zero? den)
          (none)
          (some (/ num den))))
```

```
(test (safe-div 42 0) (none))
(test (safe-div 42 12) (some 7/2))
```

Option types and Unions 2/2

p. 159

```
div2 (define (safe-div2 [num : Number] [den : Number])  
      (if (zero? den)  
          #f  
          (/ num den)))
```

```
(test (safe-div2 42 0) #f)  
(test (safe-div2 42 12) 7/2)
```

- ▶ What is the type of `safe-div2`
- ▶ Arguably cleaner code than the `Option` case.
- ▶ Notice more annotation needed

Polymorphic return values 1/2

Recall our (plait-style) rule for `if`:

$$\frac{\Gamma \vdash c : \text{Bool} \quad \Gamma \vdash T : W \quad \Gamma \vdash E : W}{\Gamma \vdash (\text{if } c \ T \ E) : W}$$

With union types we can generalize to

$$\frac{\Gamma \vdash c : \text{Bool} \quad \Gamma \vdash T : W \quad \Gamma \vdash E : Z}{\Gamma \vdash (\text{if } c \ T \ E) : (U \ W \ Z)}$$

Polymorphic return values 2/2

Looking at `if / cond` in functions we get

plus

```
(define (plus l r)
  (cond
    [(and (number? l) (number? r)) (+ l r)]
    [(and (string? l) (string? r)) (string-append l r)]
    [else (error 'plus "mismatch")]))
(test (plus 6 7) 13)
(test (plus "hello" " world") "hello world")
```

- ▶ what is the type of `plus`?
- ▶ why don't the parameters need annotation
- ▶ should we anyway?