Functional Programming 2: Defining Data Types

CS 350

Dr. Joseph Eremondi

Last updated: July 3, 2024

Warmup: Pairs

Pairs: “AND” for types

- `(Number * Boolean)`

Pairs: “AND” for types

- `(Number * Boolean)`
    - Cartesian product

Pairs: “AND” for types

- `(Number * Boolean)`
  - Cartesian product
    - “AND” for types

Pairs: “AND” for types

- `(Number * Boolean)`
  - Cartesian product
    - “AND” for types
  - A value of this type contains a `Number` AND a `Boolean`

Pairs: "AND" for types

- `(Number * Boolean)`
  - Cartesian product
    - "AND" for types
  - A value of this type contains a `Number` AND a `Boolean`
  - Why is it infix? Who knows

Pairs: “AND” for types

- `(Number * Boolean)`
  - Cartesian product
    - “AND” for types
  - A value of this type contains a `Number` AND a `Boolean`
  - Why is it infix? Who knows
    - Maybe to help distinguish it from multiplication

Pairs: “AND” for types

- `(Number * Boolean)`
  - Cartesian product
    - “AND” for types
  - A value of this type contains a `Number` AND a `Boolean`
  - Why is it infix? Who knows
    - Maybe to help distinguish it from multiplication
- Projections

Pairs: “AND” for types

- `(Number * Boolean)`
  - Cartesian product
    - “AND” for types
  - A value of this type contains a `Number` AND a `Boolean`
  - Why is it infix? Who knows
    - Maybe to help distinguish it from multiplication
- Projections
  - Get the data from the pairs

Pairs: “AND” for types

- `(Number * Boolean)`
    - Cartesian product
        - “AND” for types
    - A value of this type contains a `Number` AND a `Boolean`
    - Why is it infix? Who knows
        - Maybe to help distinguish it from multiplication
- Projections
    - Get the data from the pairs

Pairs: “AND” for types

- `(Number * Boolean)`
  - Cartesian product
    - “AND” for types
  - A value of this type contains a `Number` AND a `Boolean`
  - Why is it infix? Who knows
    - Maybe to help distinguish it from multiplication
- Projections
  - Get the data from the pairs

```
(define myPair : (Number * Boolean)
   (pair 2 #t))
(fst myPair) ;;Number
(snd myPair) ;;Boolean
```

Pairs: "AND" for types

- `(Number * Boolean)`
  - Cartesian product
    - "AND" for types
  - A value of this type contains a `Number` AND a `Boolean`
  - Why is it infix? Who knows
    - Maybe to help distinguish it from multiplication
- Projections
  - Get the data from the pairs

```
(define myPair : (Number * Boolean)
   (pair 2 #t))
(fst myPair) ;;Number
(snd myPair) ;;Boolean
```

```
2
#t
```

Pairs in General

- For any types `'a` and `'b` there is a type `('a * 'b)`

Pairs in General

- For any types `’a` and `’b` there is a type `(’a * ’b)`
- Build with `pair : (’a ’b -> (’a * ’b))`

Pairs in General

- For any types `'a` and `'b` there is a type `('a * 'b)`
- Build with `pair : ('a 'b -> ('a * 'b))`
  - Takes one argument of each type, produces the pair

Pairs in General

- For any types `'a` and `'b` there is a type `('a * 'b)`
- Build with `pair : ('a 'b -> ('a * 'b))`
  - Takes one argument of each type, produces the pair
- Projections

Pairs in General

- For any types `'a` and `'b` there is a type `('a * 'b)`
- Build with `pair : ('a 'b -> ('a * 'b))`
  - Takes one argument of each type, produces the pair
- Projections
  - `fst : (('a * 'b) -> 'a)`

Pairs in General

- For any types `'a` and `'b` there is a type `('a * 'b)`
- Build with `pair : ('a 'b -> ('a * 'b))`
    - Takes one argument of each type, produces the pair
- Projections
    - `fst : (('a * 'b) -> 'a)`
    - `snd : (('a * 'b) -> 'b)`

Algebraic Data Types

“OR” for types

- Pairs gave us “AND”

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?
  - `(OR Number Boolean)` should be the type of values that are either a number or a boolean

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?
  - `(OR Number Boolean)` should be the type of values that are either a number or a boolean
  - Want a tag so we can check which one it is

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?
  - `(OR Number Boolean)` should be the type of values that are either a number or a boolean
  - Want a tag so we can check which one it is
    - Called a “constructor”

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?
  - `(OR Number Boolean)` should be the type of values that are either a number or a boolean
  - Want a tag so we can check which one it is
    - Called a “constructor”
    - not the same as Java/OOP constructor

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?
  - `(OR Number Boolean)` should be the type of values that are either a number or a boolean
  - Want a tag so we can check which one it is
    - Called a “constructor”
    - not the same as Java/OOP constructor
- Saw some examples like this already

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?
  - `(OR Number Boolean)` should be the type of values that are either a number or a boolean
  - Want a tag so we can check which one it is
    - Called a “constructor”
    - not the same as Java/OOP constructor
- Saw some examples like this already
  - A number is zero OR one plus another number

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?
  - `(OR Number Boolean)` should be the type of values that are either a number or a boolean
  - Want a tag so we can check which one it is
    - Called a “constructor”
    - not the same as Java/OOP constructor
- Saw some examples like this already
  - A number is zero OR one plus another number
  - A list is empty OR an element cons-ed to another list

“OR” for types

- Pairs gave us “AND”
- What does “OR” look like for types?
  - `(OR Number Boolean)` should be the type of values that are either a number or a boolean
  - Want a tag so we can check which one it is
    - Called a “constructor”
    - not the same as Java/OOP constructor
- Saw some examples like this already
  - A number is zero OR one plus another number
  - A list is empty OR an element cons-ed to another list
- Racket lets us define our own types mixing AND and OR

First example

First example

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- Shape is a *datatype*

First example

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- Shape is a *datatype*
- It has two *constructors*, `Rectangle` and `Circle`

First example

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- `Shape` is a *datatype*
- It has two *constructors*, `Rectangle` and `Circle`
    - i.e. a Shape is a circle or a rectangle

First example

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- `Shape` is a *datatype*
- It has two *constructors*, `Rectangle` and `Circle`
    - i.e. a Shape is a circle or a rectangle
- `Rectangle` has two fields with type Number, `length` and `width`

First example

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- `Shape` is a *datatype*
- It has two *constructors*, `Rectangle` and `Circle`
  - i.e. a Shape is a circle or a rectangle
- `Rectangle` has two fields with type Number, `length` and `width`
- `Circle` has one field with type Number

Creating values of a datatype

Creating values of a datatype

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- We construct a Shape by calling a constructor

Creating values of a datatype

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- We construct a Shape by calling a constructor
  - Doesn't **do** anything except package the data together

Creating values of a datatype

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- We construct a Shape by calling a constructor
  - Doesn't **do** anything except package the data together
- A Shape either has two numbers OR one number

Creating values of a datatype

```
(define-type Shape
  (Rectangle [length : Number]
             [width : Number])
  (Circle [radius : Number]))

(define tv (Rectangle 16 9))
(define loonie (Circle 1))
```

- We construct a Shape by calling a constructor
    - Doesn't **do** anything except package the data together
- A Shape either has two numbers OR one number
    - Depending on the tag

Auto-generated Functions

Auto-generated Functions

```
(Rectangle? tv)
(Circle? tv)
(Rectangle-length tv)
;; (test/exn (Circle-radius tv) "")
  ;;raises an error, no such field present
```

Pattern matching

- Don't want to accidentally get a field that doesn't exist

Pattern matching

- Don't want to accidentally get a field that doesn't exist
- Almost always want to use the fields in the solution

Pattern matching

- Don’t want to accidentally get a field that doesn’t exist
- Almost always want to use the fields in the solution
- Solution: pattern-matching

Pattern matching

- Don’t want to accidentally get a field that doesn’t exist
- Almost always want to use the fields in the solution
- Solution: pattern-matching

Pattern matching

- Don’t want to accidentally get a field that doesn’t exist
- Almost always want to use the fields in the solution
- Solution: pattern-matching

```
(define (area [shp : Shape]) : Number
  (type-case Shape shp
     [(Rectangle l w)
       (* l w)]
     [(Circle r)
       (* 3.14 (* r r))])
)
(area tv)
(area loonie)
```

Pattern matching

- Don't want to accidentally get a field that doesn't exist
- Almost always want to use the fields in the solution
- Solution: pattern-matching

```
(define (area [shp : Shape]) : Number
  (type-case Shape shp
     [(Rectangle l w)
       (* l w)]
     [(Circle r)
       (* 3.14 (* r r))])
)
(area tv)
(area loonie)
```

```
144
3.14
```

Total Matching

- Missing a case in pattern matching is a *syntax error*

Total Matching

- Missing a case in pattern matching is a *syntax error*
- Lets us know we are safe

Total Matching

- Missing a case in pattern matching is a *syntax error*
- Lets us know we are safe
- Can add an else clause to handle multiple cases

Total Matching

- Missing a case in pattern matching is a *syntax error*
- Lets us know we are safe
- Can add an else clause to handle multiple cases
    - See Racket window

E.g. Representing Failure

- In plait standard library

E.g. Representing Failure

- In plait standard library

E.g. Representing Failure

- In plait standard library

```
(define-type (Optionof 'a)
  (none)
  (some [v : 'a]))
```

- Pattern matching guarantees no null pointer errors

E.g. Representing Failure

- In plait standard library

```
(define-type (Optionof 'a)
  (none)
  (some [v : 'a]))
```

- Pattern matching guarantees no null pointer errors
    - We'll see a more detailed example

Most Generic Form

Most Generic Form

```
(define-type (Either 'a 'b)
  (Left [inLeft : 'a])
  (Right [inRight : 'b])
)
```

- You can define all (non-recursive) datatypes with this and pairs

Most Generic Form

```
(define-type (Either 'a 'b)
  (Left [inLeft : 'a])
  (Right [inRight : 'b])
)
```

- You can define all (non-recursive) datatypes with this and pairs
- e.g. Shape is `(Either (Number * Number) Number)`

Recursive Data

Self-reference in types

- The real power of datatypes is the ability to have fields of the type being defined

Self-reference in types

- The real power of datatypes is the ability to have fields of the type being defined
- This allows us to define **trees**

Self-reference in types

- The real power of datatypes is the ability to have fields of the type being defined
- This allows us to define **trees**
    - of arbitrary depth

Self-reference in types

- The real power of datatypes is the ability to have fields of the type being defined
- This allows us to define **trees**
    - of arbitrary depth
- Data can be traversed using recursion

Example: Lists as a dataype

Example: Lists as a dataype

```
(define-type NumList
  (Nil)
  (Cons [head : Number] [tail : NumList]))
```

- Note: this is not quite how lists are defined in Racket/plait

Example: Lists as a dataype

```
(define-type NumList
  (Nil)
  (Cons [head : Number] [tail : NumList]))
```

- Note: this is not quite how lists are defined in Racket/plait
  - But they could be!

Example: Lists as a dataype

```
(define-type NumList
  (Nil)
  (Cons [head : Number] [tail : NumList]))
```

- Note: this is not quite how lists are defined in Racket/plait
    - But they could be!
- Recursive fields in datatype $\rightarrow$ recursive calls in template

Example: Lists as a dataype

```
(define-type NumList
  (Nil)
  (Cons [head : Number] [tail : NumList]))
```

- Note: this is not quite how lists are defined in Racket/plait
  - But they could be!
- Recursive fields in datatype $\rightarrow$ recursive calls in template

Example: Lists as a dataype

```
(define-type NumList
  (Nil)
  (Cons [head : Number] [tail : NumList]))
```

- Note: this is not quite how lists are defined in Racket/plait
  - But they could be!
- Recursive fields in datatype $\rightarrow$ recursive calls in template

```
(define (sum [xs : NumList])
  (type-case NumList xs
             [(Nil)
              0]
             [(Cons h t)
              (let ([sumRest (sum t)])
                    (+ h (sum t)))
              ]))
(sum (Cons 100 (Cons 20 (Cons 3 (Nil)))))
```

Example: Filesystem

- Model of a file system

Example: Filesystem

- Model of a file system
  - Not how is implemented on disk

Example: Filesystem

- Model of a file system
  - Not how is implemented on disk

Example: Filesystem

- Model of a file system
    - Not how is implemented on disk

```
(define-type Filesystem
  (File [name : String]
        [data : Number])
  (Folder [name : String]
          [contents : (Listof Filesystem)]))
```

Linear Search using Recursion

- Find the first matching file

Linear Search using Recursion

- Find the first matching file

Linear Search using Recursion

- Find the first matching file

```
(define (search [target : String]
                [fs : Filesystem]) : (Optionof Number)
  (type-case Filesystem fs
             [(File name data)
              (if (string=? name target)
                  (some data)
                  (none))]
             [(Folder _ contents)
               (searchList target contents)]))
```

Helper Function: Searching the list

- We have mutually recursive types

Helper Function: Searching the list

- We have mutually recursive types
  - `Filesystem` contains `(Listof Filesystem)`

Helper Function: Searching the list

- We have mutually recursive types
  - `Filesystem` contains `(Listof Filesystem)`
  - `(Listof Filesystem)` contains `Filesystem`

Helper Function: Searching the list

- We have mutually recursive types
  - `Filesystem` contains `(Listof Filesystem)`
  - `(Listof Filesystem)` contains `Filesystem`
- So we use mutually-recursive functions

Helper Function: Searching the list

- We have mutually recursive types
  - `Filesystem` contains `(Listof Filesystem)`
  - `(Listof Filesystem)` contains `Filesystem`
- So we use mutually-recursive functions

Helper Function: Searching the list

- We have mutually recursive types
  - `Filesystem` contains `(Listof Filesystem)`
  - `(Listof Filesystem)` contains `Filesystem`
- So we use mutually-recursive functions

```
(define (searchList [target : String]
                    [files : (Listof Filesystem)])
        : (Optionof Number)
  (type-case (Listof Filesystem) files
             [empty (none)]
             [(cons h t)
              (let ([result (search target h)])
                (if (none? result)
                    (searchList target t)
                    result))
              ]))
```

Testing the search

Testing the search

```
(define InnerSolarSystem
  (Folder "Sun"
          (list (File "Mercury" 1)
                (File "Venus" 2)
                (Folder "EarthSystem"
                        (list (File "Earth" 3)
                              (File "Moon" 3.5)))
                (Folder "MarsSystem"
                        (list (File "Mars" 4)
                              (File "Phobos" 4.3)
                              (File "Diemos" 4.6))))))

(search "Moon" InnerSolarSystem)
(search "Jupiter" InnerSolarSystem)
```

Testing the search

```
(define InnerSolarSystem
  (Folder "Sun"
          (list (File "Mercury" 1)
                (File "Venus" 2)
                (Folder "EarthSystem"
                        (list (File "Earth" 3)
                              (File "Moon" 3.5)))
                (Folder "MarsSystem"
                        (list (File "Mars" 4)
                              (File "Phobos" 4.3)
                              (File "Diemos" 4.6))))))

(search "Moon" InnerSolarSystem)
(search "Jupiter" InnerSolarSystem)
```

```
(some 3.5)
(none)
```