Data structures

The INFDEV@HR Team

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Lecture topics

- Let
- Tuples
- Discriminated unions (polymorphism)

Let-in

Let-in

Idea

- Sometimes we wish to give a name to a value or a computation, to reuse later
- This construct is called `let-in`
- We could then say something like `let age = 9 in age + age`
- We can nest `let-in` constructs, and then say something like `let age = 9 in (let x = 2 in age * x)`

Let-in

Idea

- Sometimes we wish to give a name to a value or a computation, to reuse later
- This construct is called `let-in`
- We could then say something like `let age = 9 in age + age`
- We can nest `let-in` constructs, and then say something like `let age = 9 in (let x = 2 in age * x)`
- This makes code significantly more readable, as it looks like a series of declarations top-to-bottom

Let-in


```
let age = 9 in (age + age)
```

Let-in


```
let age = 9 in (age + age)
```

```
let age = 9 in (age + age)
```

Let-in


```
let age = 9 in (age + age)
```

Let-in


```
let age = 9 in (age + age)
```

```
((λage→(age + age)) 9)
```

Let-in


```
((λage→(age + age)) 9)
```

Let-in


```
((λage→(age + age)) 9)
```

```
((λage→(age + age)) 9)
```

Let-in


```
((λage→(age + age)) 9)
```

Let-in


```
((λage→(age + age)) 9)
```

```
( 9  +  9 )
```

Let-in


```
(9 + 9)
```

Let-in


```
(9 + 9)
```

```
(9 + 9)
```

Let-in


```
(9 + 9)
```

Let-in


```
(9 + 9)
```

```
18
```

Data types

Data types

Overview

- We now move on to ways to define data types
- The definitions will be both **minimal** and **composable**
- Classes, polymorphism, etc. can all be rendered under our definitions, so we miss nothing substantial

Data types

Overview

Notice: from now on we will start ignoring the reduction steps for simple terms such as 3+3, `x = 0`, etc. for brevity

Data types

Minimality

- The lambda calculus has so far proven very powerful, despite its size
- We do not need hundreds of different operators, we can simply build them[a]
- The only extension needed is purely syntactic in nature to make it more mnemonic, but this is only skin-deep and requires no change to the underlying mechanisms of the lambda calculus

[a] and more

Data types

Minimality

- In defining data types we wish to maintain this minimality
- We do not want dozens of separate, competing data types all slightly overlapping

Data types

Fundamental scenarios

- **Tuples**: storing multiple things together at the same time, like the fields and methods in a class
- **Unions**: storing either one of various things at a time, like an interface that is exactly one of its concrete implementors

Data types

The importance of composition

- We just need to cover the case of two items, higher numbers come through composition
- For example, given the ability to store a pair, we can build a pair of pairs to create arbitrary tuples
- Similarly, given the ability to store either of two values, we can build either of many values with nesting

Tuples

Tuples

- A pair of values is defined simply as something that stores these two values
- We can extract them by giving the pair a function that will receive the values

```
(λx y→ (λf→((f x) y)))
```

Tuples


```
(1, 2)
```

Tuples


```
(1, 2)
```

```
((,) 1) 2)
```

Tuples


```
(((,) 1) 2)
```

Tuples


```
(((,) 1) 2)
```

```
((λx y→ (λf→((f x) y))) 1) 2)
```

Tuples


```
(((λx y→ (λf→((f x) y))) 1) 2)
```

Tuples


```
(((λx y→ (λf→((f x) y))) 1) 2)
```

```
(((λx y→ (λf→((f x) y))) 1) 2)
```

Tuples


```
(((λx y→ (λf→((f x) y))) 1) 2)
```

Tuples


```
(((λx y→ (λf→((f x) y))) 1) 2)
```

```
((λy f→((f 1) y)) 2)
```

Tuples


```
((λy f→((f 1) y)) 2)
```

Tuples


```
((λy f→((f 1) y)) 2)
```

```
((λy f→((f 1) y)) 2)
```

Tuples


```
((λy f→((f 1) y)) 2)
```

Tuples


```
((λy f→((f 1) y)) 2)
```

```
(λf→((f 1) 2))
```

Tuples

- We can define two utility functions that, given a pair, extract the first or second value
- They are usually called π_1 and π_2, or `fst` and `snd`

```
(λp→(p (λx y→x)))
```

```
(λp→(p (λx y→y)))
```

Tuples


```
(π1 (1, 2))
```

Tuples


```
(π1 (1, 2))
```

```
(π1 (1, 2))
```

Tuples


```
(π1 (1, 2))
```

Tuples


```
(π1 (1, 2))
```

```
((λp→(p (λx y→x))) (1, 2))
```

Tuples


```
((λp→(p (λx y→x))) (1, 2))
```

Tuples


```
((λp→(p (λx y→x))) (1, 2))
```

```
((λp→(p (λx y→x))) (((,) 1) 2))
```

Tuples


```
((λp→(p (λx y→x))) (((,) 1) 2))
```

Tuples


```
((λp→(p (λx y→x))) (((,) 1) 2))
```

```
((λp→(p (λx y→x))) ((λx y→ (λf→((f x) y))) 1)
    2))
```

Tuples


```
((λp→(p (λx y→x))) (((λx y→ (λf→((f x) y)))
    1) 2))
```

Tuples


```
((λp→(p (λx y→x))) (((λx y→ (λf→((f x) y))) 1) 2))
```

```
((λp→(p (λx y→x))) (((λx y→ (λf→((f x) y))) 1) 2))
```

Tuples


```
((λp→(p (λx y→x))) (((λx y→ (λf→((f x) y))) 1)
    2))
```

Tuples


```
((λp→(p (λx y→x))) (((λx y→ (λf→((f x) y))) 1)
    2))
```

```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

Tuples


```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

Tuples


```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

Tuples


```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

Tuples


```
((λp→(p (λx y→x))) ((λy f→((f 1) y)) 2))
```

```
( ((λy f→((f 1) y)) 2) (λx y→x))
```

Tuples


```
(((λy f→((f 1) y)) 2) (λx y→x))
```

Tuples


```
(((λy f→((f 1) y)) 2) (λx y→x))
```

```
(((λy f→((f 1) y)) 2) (λx y→x))
```

Tuples


```
(((λy f→((f 1) y)) 2) (λx y→x))
```

Tuples


```
(((λy f→((f 1) y)) 2) (λx y→x))
```

```
((λf→((f 1) 2)) (λx y→x))
```

Tuples


```
((λf→((f 1) 2)) (λx y→x))
```

Tuples


```
((λf→((f 1) 2)) (λx y→x))
```

```
((λf→((f 1) 2)) (λx y→x))
```

Tuples


```
((λf→((f 1) 2)) (λx y→x))
```

Tuples


```
((λf→((f 1) 2)) (λx y→x))
```

```
(( (λx y→x)  1)  2)
```

Tuples


```
(((λx y→x) 1) 2)
```

Tuples


```
(((λx y→x) 1) 2)
```

```
(((λx y→x) 1) 2)
```

Tuples


```
(((λx y→x) 1) 2)
```

Tuples


```
(((λx y→x) 1) 2)
```

```
((λy→ 1 ) 2)
```

Tuples


```
((λy→1)  2)
```

Tuples


```
((λy→1) 2)
```

```
((λy→1) 2)
```

Tuples


```
((λy→1) 2)
```

Tuples


```
((λy→1) 2)
```

```
1
```

Tuples

Pair of values

We should expect that π_1 and π_2 are inverse operations to constructing a pair, as they destroy it

Tuples

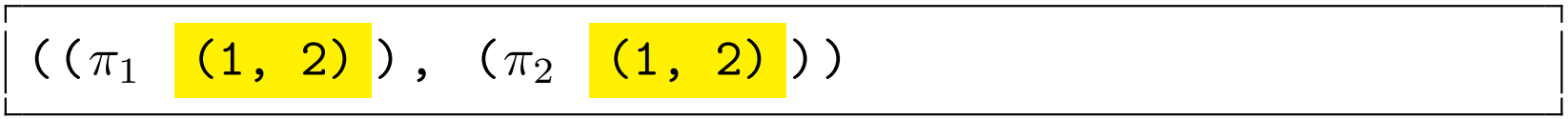

```
let p = (1, 2) in ((π1 p), (π2 p))
```

Tuples


```
let p = (1, 2) in ((π1 p), (π2 p))
```

```
((π1 (1, 2)), (π2 (1, 2)))
```

Tuples



Tuples


```
((π1 (1, 2)), (π2 (1, 2)))
```

```
((π1 (1, 2)), (π2 (1, 2)))
```

Tuples


```
((π1 (1, 2)), (π2 (1, 2)))
```

Tuples


```
((π1 (1, 2)), (π2 (1, 2)))
```

```
( 1 , (π2 (1, 2)))
```

Tuples


```
(1, (π2 (1, 2)))
```

Tuples


```
(1, (π2 (1, 2)))
```

```
(1, (π2 (1, 2)))
```

Tuples

(1, $\underline{(\pi_2\ (1,\ 2))}$)

Tuples


```
(1, (π2 (1, 2)))
```

```
(1, 2)
```

Tuples


```
(1, 2)
```

Tuples


```
(1, 2)
```

```
(1, 2)
```

Discriminated unions

Discriminated unions

- A choice of values is defined simply as something that stores either of two possible values
- We call such a choice a **discriminated union**
- We build a discriminated union with either of two functions to build the first or the second value
- They are usually called `inl` and `inr`[a]

[a] *in* stands for injection, and *l* and *r* stand for left and right

```
(λx→ (λf g→(f x)))
```

```
(λy→ (λf g→(g y)))
```

Discriminated unions


```
(inl 1)
```

Discriminated unions


```
(inl 1)
```

```
(inl 1)
```

Discriminated unions


```
(inl 1)
```

Discriminated unions


```
(inl 1)
```

```
((λx→ (λf g→(f x))) 1)
```

Discriminated unions


```
((λx→ (λf g→(f x))) 1)
```

Discriminated unions


```
((λx→ (λf g→(f x))) 1)
```

```
((λx→ (λf g→(f x))) 1)
```

Discriminated unions


```
((λx→ (λf g→(f x))) 1)
```

Discriminated unions


```
((λx→ (λf g→(f x))) 1)
```

```
(λf g→(f 1))
```

Discriminated unions


```
(λf g→(f 1))
```

Discriminated unions


```
(λf g→(f 1))
```

```
(λf g→(f 1))
```

Discriminated unions


```
(λf g→(f 1))
```

Discriminated unions


```
(λf g→(f 1))
```

```
(inl 1)
```

Discriminated unions


```
(inr TRUE)
```

Discriminated unions


```
(inr TRUE)
```

```
(inr TRUE)
```

Discriminated unions


```
(inr TRUE)
```

Discriminated unions


```
(inr TRUE)
```

```
( (λy→ (λf g→(g y))) TRUE)
```

Discriminated unions


```
((λy→ (λf g→(g y))) TRUE)
```

Discriminated unions


```
((λy→ (λf g→(g y))) TRUE)
```

```
((λy→ (λf g→(g y))) TRUE)
```

Discriminated unions


```
((λy→ (λf g→(g y))) TRUE)
```

Discriminated unions


```
((λy→ (λf g→(g y))) TRUE)
```

```
(λf g→(g TRUE))
```

Discriminated unions


```
(λf g→(g TRUE))
```

Discriminated unions


```
(λf g→(g TRUE))
```

```
(λf g→(g TRUE))
```

Discriminated unions


```
(λf g→(g TRUE))
```

Discriminated unions


```
(λf g→(g TRUE))
```

```
(inr TRUE)
```

Discriminated unions

- Extracting the input of a discriminated union is a process known as `match`[a]
- Given a union and two functions (one per case), if the union was the first case we apply the first function, otherwise we apply the second function

[a] which is a sort of `switch`, just on steroids

```
(λu→ (λf g→((u f) g)))
```

Discriminated unions


```
(((match (inl 1)) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

Discriminated unions


```
(((match (inl 1)) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

```
(((match (inl 1)) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

Discriminated unions


```
(((match (inl 1)) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

Discriminated unions


```
(((match (inl 1)) (λx→(x + 1))) (λy→(y ∧
    FALSE)))
```

```
(((  (λu→ (λf g→((u f) g)))  (inl 1)) (λx→(x + 1)
    )) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) (inl 1)) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (
    (λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) ((λx→ (λf g→(f x))
    ) 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) ((λx→ (λf g→(f x))
    ) 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g)))
    ((λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g)))
    ((λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g)))
    ((λx→ (λf g→(f x))) 1)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx
    →(x + 1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→
    (x + 1))) (λy→(y ∧ FALSE)))
```

```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→(x +
    1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((((λu→ (λf g→((u f) g))) (λf g→(f 1))) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

```
(((λf g→((λf g→(f 1)) f) g)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
(((λf g→(((λf g→(f 1)) f) g)) (λx→(x + 1)))
    (λy→(y ∧ FALSE)))
```

Discriminated unions


```
(((λf g→(((λf g→(f 1)) f) g)) (λx→(x + 1)))
    (λy→(y ∧ FALSE)))
```

```
(((λf g→(((λf g→(f 1)) f) g)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

Discriminated unions


```
(((λf g→(((λf g→(f 1)) f) g)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

Discriminated unions


```
(((λf g→(((λf g→(f 1)) f) g)) (λx→(x + 1))) (λy→(
    y ∧ FALSE)))
```

```
((λg→(((λf g→(f 1)) (λx→(x + 1))) g)) (λy→(y
    ∧ FALSE)))
```

Discriminated unions


```
((λg→(((λf g→(f 1)) (λx→(x + 1))) g)) (λy→(
    y ∧ FALSE)))
```

Discriminated unions


```
((λg→(((λf g→(f 1)) (λx→(x + 1))) g)) (λy→(
    y ∧ FALSE)))
```

```
((λg→(((λf g→(f 1)) (λx→(x + 1))) g))
    (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((λg→(((λf g→(f 1)) (λx→(x + 1))) g))
    (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((λg→(((λf g→(f 1)) (λx→(x + 1))) g))
    (λy→(y ∧ FALSE)))
```

```
(((λf g→(f 1)) (λx→(x + 1)))
    (λy→(y ∧ FALSE)))
```

Discriminated unions


```
(((λf g→(f 1)) (λx→(x + 1))) (λy→(y ∧ FALSE))
    ))
```

Discriminated unions


```
(((λf g→(f 1)) (λx→(x + 1))) (λy→(y ∧ FALSE))
    ))
```

```
(((λf g→(f 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
(((λf g→(f 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
(((λf g→(f 1)) (λx→(x + 1))) (λy→(y ∧ FALSE)))
```

```
((λg→( (λx→(x + 1)) 1)) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((λg→((λx→(x + 1)) 1)) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((λg→((λx→(x + 1)) 1)) (λy→(y ∧ FALSE)))
```

```
((λg→((λx→(x + 1)) 1)) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((λg→((λx→(x + 1)) 1)) (λy→(y ∧ FALSE)))
```

Discriminated unions


```
((λg→((λx→(x + 1)) 1)) (λy→(y ∧ FALSE)))
```

```
((λx→(x + 1)) 1)
```

Discriminated unions


```
((λx→(x + 1)) 1)
```

Discriminated unions


```
((λx→(x + 1)) 1)
```

```
((λx→(x + 1)) 1)
```

Discriminated unions


```
((λx→(x + 1)) 1)
```

Discriminated unions


```
((λx→(x + 1)) 1)
```

```
( 1 + 1)
```

Discriminated unions


```
(1 + 1)
```

Discriminated unions


```
(1 + 1)
```

```
(1 + 1)
```

Discriminated unions


```
(1 + 1)
```

Discriminated unions


```
(1 + 1)
```

```
2
```

Discriminated unions

Choice between a pair of values

We should expect that `inl` and `inr` are inverse operations to `match`

Discriminated unions


```
(((match (inl 1)) inl) inr)
```

Discriminated unions


```
(((match (inl 1)) inl) inr)
```

```
(((match (inl 1)) inl) inr)
```

Discriminated unions


```
(((match (inl 1)) inl) inr)
```

Discriminated unions


```
(((match (inl 1)) inl) inr)
```

```
(inl 1)
```

Discriminated unions


```
(((match (inr TRUE)) inl) inr)
```

Discriminated unions


```
(((match (inr TRUE)) inl) inr)
```

```
(((match (inr TRUE)) inl) inr)
```

Discriminated unions


```
(((match (inr TRUE)) inl) inr)
```

Discriminated unions


```
(((match (inr TRUE)) inl) inr)
```

```
(inr TRUE)
```

Discriminated unions


```
(inr TRUE)
```

Discriminated unions


```
(inr TRUE)
```

```
(inr TRUE)
```

Discriminated unions


```
(inr TRUE)
```

Discriminated unions


```
(inr TRUE)
```

```
(inr (λt f→t))
```

Conclusion

Conclusion

Recap

- Lambda terms can be used to encode arbitrary basic data types
- The terms are always lambda expression which, when they get parameters passed in, identify themselves somehow
- Identification can be done by applying something (possibly even a given number of times), or returning one of the parameters

Conclusion

Recap

- The data types we have seen cover an impressive range of applications
- Tuples cover grouping data together (like the fields of a class)
- Unions cover choosing different things (like the polymorphism of an interface that might be implemented by various concrete classes)
- Combining these two covers all possible programming needs, even for more complex data structures

This is it!

The best of luck, and thanks for the attention!