Foundations of Computer Science:
Datatypes and Trees
Lecture 6

Anil Madhavapeddy
18th October 2024

Your class rep is: aag70! Get Involved :)
Custom Types

Exceptions

Recursive Types
Custom Types
Custom Types

• So far, our types have been basic: \texttt{int}, \texttt{float} or \texttt{bool} types that are built into OCaml.

• In this lecture we introduce one of the coolest features of ML-style languages in the form of custom datatypes!

• We continue to improve the abstraction of our data away from the details of its representation.
Let’s describe a vehicle

```haskell
# let number_of_wheels = function
   "bike" -> 2
| "motorbike" -> 2
| "car" -> 4
| "lorry" -> 18
```
Let’s describe a vehicle

```ocaml
# let number_of_wheels = function
  "bike" -> 2
  | "motorbike" -> 2
  | "car" -> 4
  | "lorry" -> 18

# number_of_wheels "bike"
- : int = 2

# number_of_wheels "motorbike"
???
```
Let's describe a vehicle

```
# let number_of_wheels = function
  "bike" -> 2
  | "motorbike" -> 2
  | "car" -> 4
  | "lorry" -> 18

# number_of_wheels "bike"
- : int = 2

# number_of_wheels "Motorbike"
???
```
Let’s describe a vehicle

```ocaml
# let number_of_wheels = function
    “bike” -> 2
  | “motorbike” -> 2
  | “car” -> 4
  | “lorry” -> 18
```

```ocaml
# number_of_wheels “bike”
- : int = 2

# number_of_wheels “motorbike”
???
```

How can we make illegal states unrepresentable?
An Enumeration Type

```plaintext
# type vehicle =
| Bike
| Motorbike
| Car
| Lorry
```
An Enumeration Type

- We have declared a new type `vehicle`
- Instead of representing any string, it can only contain the four constants defined.
- These four constants become the constructors of the `vehicle` type.
An Enumeration Type

- The *representation* in memory is more efficient than using strings.
- Adding new types of vehicles is straightforward by extending the definitions.
- Different custom types cannot be intermixed, unlike strings or integers.
Declaring functions on vehicles

```ocaml
# let wheels = function
| Bike  -> 2
| Motorbike  -> 2
| Car  -> 4
| Lorry  -> 18

val wheels : vehicle -> int = <fun>
```
Declaring functions on vehicles

```ocaml
# let wheels = function
 | Bike    -> 2
 | Motorbike    -> 2
 | Car       -> 4
 | Lorry     -> 18
val wheels : vehicle -> int = <fun>
```

```ocaml
# let wheels = function
 | "bike"    -> 2
 | "motorbike"    -> 2
 | "car"       -> 4
 | "lorry"     -> 18
val wheels : string -> int = <fun>
```

- The *representation* in memory is more efficient than using strings.
- Different custom types cannot be intermixed, unlike strings or integers.
Declaring functions on vehicles

```ocaml
# let wheels = function
    | Bike -> 2
    | Motorbike -> 2
    | Car -> 4
    | Lorry -> 18
val wheels : vehicle -> int = <fun>
```

```ocaml
# let wheels = function
    | Bike -> 2
    | Motorbike -> 2
    | Car -> 4

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Orange
val wheels : vehicle -> int = <fun>
```

- Adding new types of vehicles is straightforward by extending the definitions and fixing warnings.
Declaring functions on vehicles

```ocaml
# type vehicle = Bike
  | Motorbike of int
  | Car of bool
  | Lorry of int
```

- OCaml generalises the notion of enumeration types to allow data to be stored alongside each variant.

```ocaml
# Bike
# Motorbike 250
# Car true
# Lorry 500
```
Declaring functions on vehicles

OCaml generalises the notion of enumeration types to allow data to be stored alongside each variant.

```ocaml
# type vehicle = Bike
  | Motorbike of int
  | Car       of bool
  | Lorry     of int
```

```ocaml
# type vehicle = Bike
  | Motorbike of int (* engine size in CCs *)
  | Car       of bool (* true if a Reliant Robin *)
  | Lorry     of int (* number of wheels *)
```
Declaring functions on vehicles

```ocaml
# type vehicle = Bike
| Motorbike of int
| Car      of bool
| Lorry    of int
```

- OCaml generalises the notion of enumeration types to allow data to be stored alongside each variant.

```ocaml
# type vehicle = Bike
| Motorbike of int (* engine size in CCs *)
| Car      of bool (* true if a Reliant Robin *)
| Lorry    of int (* number of wheels *)
```

An OCaml comment allows annotation of source code.
Declaring functions on vehicles

OCaml generalises the notion of enumeration types to allow data to be stored alongside each variant.

Even though they have different data, they are all of type vehicle when wrapped by the constructor.
A finer wheel computation

```ocaml
# let wheels = function
| Bike  -> 2
| Motorbike _ -> 2
| Car robin   -> if robin then 3 else 4
| Lorry w     -> w
```

- A Bike has two wheels.
- A Motorbike has two wheels.
- A Reliant Robin has three wheels; all other cars have four.
- A Lorry has the number of wheels stored with its constructor.
A finer wheel computation

```ocaml
# let is_reliant_robin = function
  | Car true -> true
  | _ -> false
```
Exceptions
Exceptions

• During a computation, what if something goes wrong?
  • Division by zero
  • Pattern matching failure

• Exception handling allows us to recover from these:
  • Raising an exception abandons the current expression
  • Handling the exception attempts an alternative

• Raising and handling can be separated in the source code
Exceptions

# exception Failure
exception Failure

# exception NoChange of int
exception NoChange of int

# raise Failure
Exception: Failure.

• Each exception declaration introduces a distinct type of exception that can be handled separately.

• Exceptions are like enumerations and can have data attached to them.
## Exceptions

```ocaml
# try
    print_endline "pre exception";
    raise (NoChange 1);
    print_endline "post exception";
with
    | NoChange _ ->
      print_endline "handled a NoChange exception"

Line 3, characters 5-23:
Warning 21: this statement never returns (or has an unsound type.)
pre exception
handled a NoChange exception
- : unit = ()
```

- `raise` dynamically jumps to the nearest `try/with` handler that matches that exception

- Unlike some languages, OCaml does not mark a function to indicate that an exception might be raised.
Exceptions

```
# try
print_endline "pre exception";
raise (NoChange 1);
print_endline "post exception";
with |
    | NoChange _ ->
    print_endline "handled a NoChange exception"
```

- `raise` dynamically jumps to the nearest `try/with` handler that matches that exception

- Unlike some languages, OCaml does not mark a function to indicate that an exception might be raised.
Exceptions

```ocaml
# try
  print_endline "pre exception";
  raise (NoChange 1);
  print_endline "post exception";
with
  | NoChange _ ->
    print_endline "handled a NoChange exception"
```

- raise dynamically jumps to the nearest try/with handler that matches that exception

- Unlike some languages, OCaml does not mark a function to indicate that an exception might be raised.
Exceptions

```ocaml
# try
  print_endline "pre exception";
  raise (NoChange 1);
  print_endline "post exception";
with |
      | NoChange _ ->
      | print_endline "handled a NoChange exception"
```

- `raise` dynamically jumps to the nearest `try/with` handler that matches that exception.

- Unlike some languages, OCaml does not mark a function to indicate that an exception might be raised.
let rec change till amt = 
  if amt = 0 then 
      [ [ ] ] 
  else 
      match till with 
      | [ ] -> [ ] 
      | c::till -> 
          if amt < c then 
              change till amt 
          else 
              let rec allc = function 
                  | [ ] -> [ ] 
                  | cs :: css -> (c::cs) :: allc css 
              in 
              allc (change (c::till) (amt - c)) @ 
              change till amt
Change with backtracking

```ocaml
# exception Change
let rec change till amt =
  if amt = 0 then
    []
  else
    match till with
    | [] ->
      raise Change
    | c::till ->
      if amt < 0 then
        raise Change
      else
        try
          c :: change (c::till) (amt - c)
        with Change ->
          change till amt

exception Change
val change : int list -> int -> int list = <fun>
```
# exception Change
let rec change till amt =  
  if amt = 0 then  
    []  
  else  
    match till with  
    | [] ->  
      raise Change  
    | c::till ->  
      if amt < 0 then  
        raise Change  
      else  
        try  
          c :: change (c::till) (amt - c)  
        with Change ->  
          change till amt  
exception Change
val change : int list -> int -> int list = <fun>
```ml
# exception Change
let rec change till amt =
  if amt = 0 then
    []
  else
    match till with
    | []  ->
      raise Change
    | c::till ->
      if amt < 0 then
        raise Change
      else
        try
          c :: change (c::till) (amt - c)
        with Change ->
          change till amt
exception Change
val change : int list -> int -> int list = <fun>
```
# exception Change
let rec change till amt =
  if amt = 0 then
    []
  else
    match till with
    | [] ->
      raise Change
    | c::till ->
      if amt < 0 then
        raise Change
      else
        try
          c :: change (c::till) (amt - c)
        with Change ->
          change till amt
  exception Change
val change : int list -> int -> int list = <fun>
# exception Change

```ocaml
let rec change till amt =
  if amt = 0 then []
  else
    match till with
    | [] -> raise Change
    | c::till ->
      if amt < 0 then raise Change
      else try
          c :: change (c::till) (amt - c)
        with Change -> change till amt
```

```ocaml
case change [5; 2; 6]
  → 5::change [5; 2] 1 with C -> change [2] 6
  → 5::(change [5; 2] -4) with C -> change [2] 1
  with C -> change [2] 6
  → 5::change [2] -1 with C -> change [1] 1
  with C -> change [2] 6
  → 5::change [1] 1 with C -> change [2] 6
  → change [2] 6
  → 2::change [1; 1] with C -> change [1] 6
  → 2::(change [1; 1] 2) with C -> change [1] 4
  with C -> change [1] 6
  → 2::(2::change [1; 1] 0) with C -> change [1] 2
  with C -> change [1] 4
  with C -> change [1] 6
  → 2::[2; 2] with C -> change [2] 6
  → [2; 2; 2]
```
Recursive Types
Binary Trees

# type 'a tree =
   Lf
 | Br of 'a * 'a tree * 'a tree
Binary Trees

- A data structure with multiple branching is called a **tree**.
- Trees are nearly as fundamental a structure as lists.
- Each node is either a **leaf** (empty) or a **branch** with a label and two subtrees.
Binary Trees

# type 'a tree =
  Lf
| Br of 'a * 'a tree * 'a tree

“Polymorphic” type
Binary Trees

```
# type 'a tree =
  Lf
| Br of 'a * 'a tree * 'a tree
```

```
# Br(1, Br(2, Br(4, Lf, Lf),
  Br(5, Lf, Lf)),
  Br(3, Lf, Lf))
```

“Polymorphic” type

```
int tree
```
Binary Trees & Lists

# type 'a tree =
  Lf
  | Br of 'a * 'a tree * 'a tree

# type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist

# Cons (1, Cons (2, Cons (3, Nil)))
- : int mylist
Polymorphism & Recursion

# type 'a tree =
| Lf
| Br of 'a * 'a tree * 'a tree

type shape =
| Null
| Join of shape * shape

type 'a option =
| None
| Some of 'a
Simple Operations on Trees

(* number of branch nodes *)
# let rec count = function
| Lf -> 0
| Br (v, t1, t2) -> 1 + count t1 + count t2
val count : 'a tree -> int = <fun>

(* length of longest path *)
# let rec depth = function
| Lf -> 0
| Br (v, t1, t2) -> 1 + max (depth t1) (depth t2)
val depth : 'a tree -> int = <fun>

• Use pattern matching to build expressions over trees

• The invariant \(\text{count}(t) \leq 2^{\text{depth}(t)} - 1\) holds above