Foundations of Computer Science

Enumerations and simple data types

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

```
type vehicle =
    Bike
  | Motorbike
  | ...
```

Custom types

```
exception NoChange
```

Exceptions

```
type t =
    Null
  | Join of t * t
```

Recursive types

This lecture introduces a powerful and distinctive feature of ML-style languages:

custom datatypes

With custom datatypes we can precisely describe the values used in our programs

Let's describe a vehicle

```
In[1]:  let number_of_wheels = function
          "bike" -> 2
        | "motorbike" -> 2
        | "car" -> 4
        | "lorry" -> 18
        Warning 8: this pattern-matching is not
        exhaustive. Here is an example of a case
        that is not matched: ""

Out[1]: val number_of_wheels : string -> int = <fun>

In[2]:  number_of_wheels "bike"

Out[2]: - : int = 2

In[3]:  number_of_wheels "motorbke"

   Out: Exception: Match_failure ("//toplevel//", 1, 23).
```

How can we **make illegal states unrepresentable**?

```
In[1]: let number_of_wheels = function
         "bike" -> 2
       | "motorbike" -> 2
       | "car" -> 4
       | "lorry" -> 18

In[2]:

In[3]:
```

How can we **make illegal states unrepresentable**?

Let’s describe a vehicle

```
In[1]:  let number_of_wheels = function
          "bike" -> 2
        | "motorbike" -> 2
        | "car" -> 4
        | "lorry" -> 18
        Warning 8: this pattern-matching is not
        exhaustive. Here is an example of a case
        that is not matched: ""

Out[1]: val number_of_wheels : string -> int = <fun>

In[2]:  number_of_wheels "bike"

Out[2]: - : int = 2

In[3]:  number_of_wheels "motorbke"

   Out: Exception: Match_failure ("//toplevel//", 1, 23).
```

How can we **make illegal states unrepresentable**?

```
In[1]:  let number_of_wheels = function
            "bike" -> 2
          | "motorbike" -> 2
          | "car" -> 4
          | "lorry" -> 18
        Warning 8: this pattern-matching is not
        exhaustive. Here is an example of a case
        that is not matched: ""

Out[1]: val number_of_wheels : string -> int = <fun>

In[2]:  number_of_wheels "bike"

Out[2]: - : int = 2

In[3]:  number_of_wheels "motorbke"

  Out:  Exception: Match_failure ("//toplevel//", 1, 23).
```

How can we **make illegal states unrepresentable**?

```
 In[1]: let number_of_wheels = function
            "bike" -> 2
          | "motorbike" -> 2
          | "car" -> 4
          | "lorry" -> 18
        Warning 8: this pattern-matching is not
        exhaustive. Here is an example of a case
        that is not matched: ""

Out[1]: val number_of_wheels : string -> int = <fun>

 In[2]: number_of_wheels "bike"

Out[2]: - : int = 2

 In[3]: number_of_wheels "motorbke"

   Out: Exception: Match_failure ("//toplevel//", 1, 23).
```

How can we **make illegal states unrepresentable**?

Let’s describe a vehicle

```
 In[1]: let number_of_wheels = function
          "bike" -> 2
        | "motorbike" -> 2
        | "car" -> 4
        | "lorry" -> 18
        Warning 8: this pattern-matching is not
        exhaustive. Here is an example of a case
        that is not matched: ""

Out[1]: val number_of_wheels : string -> int = <fun>

 In[2]: number_of_wheels "bike"

Out[2]: - : int = 2

 In[3]: number_of_wheels "motorbke"

   Out: Exception: Match_failure ("//toplevel//", 1, 23).
```

How can we **make illegal states unrepresentable**?

```
 In[1]:  let number_of_wheels = function
             "bike" -> 2
           | "motorbike" -> 2
           | "car" -> 4
           | "lorry" -> 18
         Warning 8: this pattern-matching is not
         exhaustive. Here is an example of a case
         that is not matched: ""

Out[1]:  val number_of_wheels : string -> int = <fun>

 In[2]:  number_of_wheels "bike"

Out[2]:  - : int = 2

 In[3]:  number_of_wheels "motorbke"

   Out:  Exception: Match_failure ("//toplevel//", 1, 23).
```

How can we **make illegal states unrepresentable**?

```
In[1]:  let number_of_wheels = function
            "bike" -> 2
          | "motorbike" -> 2
          | "car" -> 4
          | "lorry" -> 18
        Warning 8: this pattern-matching is not
        exhaustive. Here is an example of a case
        that is not matched: ""

Out[1]: val number_of_wheels : string -> int = <fun>

In[2]:  number_of_wheels "bike"

Out[2]: - : int = 2

In[3]:  number_of_wheels "motorbke"

   Out: Exception: Match_failure ("//toplevel//", 1, 23).
```

How can we **make illegal states unrepresentable**?

An Enumeration Type

```
type vehicle = Bike
             | Motorbike
             | Car
             | Lorry
```

```
type vehicle = Bike
             | Motorbike
             | Car
             | Lorry
```

We have declared a new type vehicle

```
type vehicle = Bike
             | Motorbike
             | Car
             | Lorry
```

We have declared a new type vehicle

Instead of representing any string it can only contain the four constants defined.

An Enumeration Type

```
type vehicle = Bike
             | Motorbike
             | Car
             | Lorry
```

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

```
type vehicle = Bike
             | Motorbike
             | Car
             | Lorry
```

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

The **representation** in memory is more efficient than using strings.

An Enumeration Type

```
type vehicle = Bike
             | Motorbike
             | Car
             | Lorry
```

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

The **representation** in memory is more efficient than using strings.

Adding new types of vehicles is straightforward by extending the definitions.

An Enumeration Type

```
type vehicle = Bike
             | Motorbike
             | Car
             | Lorry
```

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

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

```
In[4]:  let wheels = function
        | Bike -> 2
        | Motorbike -> 2
        | Car -> 4
        | Lorry -> 18

Out[4]: val wheels : vehicle -> int = <fun>

In[5]:  let wheels = function
        | "bike" -> 2
        | "motorbike" -> 2
        | "car" -> 4
        | "lorry" -> 18

Out[5]: val wheels : string -> int = <fun>
```

Declaring functions on vehicles

```
In[4]: let wheels = function
       | Bike -> 2
       | Motorbike -> 2
       | Car -> 4
       | Lorry -> 18

Out[4]: val wheels : vehicle -> int = <fun>

In[5]: let wheels = function
       | "bike" -> 2
       | "motorbike" -> 2
       | "car" -> 4
       | "lorry" -> 18

Out[5]: val wheels : string -> int = <fun>
```

Declaring functions on vehicles

```
In[4]:  let wheels = function
        | Bike -> 2
        | Motorbike -> 2
        | Car -> 4
        | Lorry -> 18

Out[4]: val wheels : vehicle -> int = <fun>
```

```
In[5]:  let wheels = function
        | "bike" -> 2
        | "motorbike" -> 2
        | "car" -> 4
        | "lorry" -> 18

Out[5]: val wheels : string -> int = <fun>
```

Declaring functions on vehicles

```
 In[4]: let wheels = function
        | Bike -> 2
        | Motorbike -> 2
        | Car -> 4
        | Lorry -> 18

Out[4]: val wheels : vehicle -> int = <fun>

 In[5]: let wheels = function
        | "bike" -> 2
        | "motorbike" -> 2
        | "car" -> 4
        | "lorry" -> 18

Out[5]: val wheels : string -> int = <fun>
```

```
In[4]:  let wheels = function
        | Bike -> 2
        | Motorbike -> 2
        | Car -> 4
        | Lorry -> 18
Out[4]: val wheels : vehicle -> int = <fun>
In[5]:  let wheels = function
        | "bike" -> 2
        | "motorbike" -> 2
        | "car" -> 4
        | "lorry" -> 18
Out[5]: val wheels : string -> int = <fun>
```

Declaring functions on vehicles

Adding new vehicle types is straightforward: extend the definitions and **fix warnings**.

```
In[6]:  let wheels = function
        | Bike -> 2
        | Motorbike -> 2
        | Car -> 4
        | Lorry -> 18

Out[6]: val wheels : vehicle -> int = <fun>

In[7]:  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:
        Lorry

Out[7]: val wheels : vehicle -> int = <fun>
```

Declaring functions on vehicles

Adding new vehicle types is straightforward: extend the definitions and **fix warnings**.

In[6]:
```
let wheels = function
| Bike -> 2
| Motorbike -> 2
| Car -> 4
| Lorry -> 18
```

Out[6]:
```
val wheels : vehicle -> int = <fun>
```

In[7]:
```
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:
Lorry
```

Out[7]:
```
val wheels : vehicle -> int = <fun>
```

Declaring functions on vehicles

Adding new vehicle types is straightforward: extend the definitions and **fix warnings**.

In[6]:
```
let wheels = function
| Bike -> 2
| Motorbike -> 2
| Car -> 4
| Lorry -> 18
```

Out[6]:
```
val wheels : vehicle -> int = <fun>
```

In[7]:
```
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:
Lorry
```

Out[7]:
```
val wheels : vehicle -> int = <fun>
```

Declaring functions on vehicles

Adding new vehicle types is straightforward: extend the definitions and **fix warnings**.

In[6]:
```
let wheels = function
| Bike -> 2
| Motorbike -> 2
| Car -> 4
| Lorry -> 18
```

Out[6]:
```
val wheels : vehicle -> int = <fun>
```

In[7]:
```
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:
Lorry
```

Out[7]:
```
val wheels : vehicle -> int = <fun>
```

Declaring functions on vehicles

Adding new vehicle types is straightforward: extend the definitions and **fix warnings**.

```
 In[6]: let wheels = function
        | Bike -> 2
        | Motorbike -> 2
        | Car -> 4
        | Lorry -> 18

Out[6]: val wheels : vehicle -> int = <fun>

 In[7]: 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:
        Lorry

Out[7]: val wheels : vehicle -> int = <fun>
```

Adding new vehicle types is straightforward: extend the definitions and **fix warnings**.

In[6]:
```
let wheels = function
| Bike -> 2
| Motorbike -> 2
| Car -> 4
| Lorry -> 18
```
Out[6]:
```
val wheels : vehicle -> int = <fun>
```
In[7]:
```
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:
Lorry
```
Out[7]:
```
val wheels : vehicle -> int = <fun>
```

Declaring functions on vehicles

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

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

```
 In[8]: Bike
Out[8]: - : vehicle = Bike
 In[9]: Motorbike 25
Out[9]: - : vehicle = Motorbike 25
In[10]: Car true
Out[10]: - : vehicle = Car true
```

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

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

```
 In[8]: Bike
Out[8]: - : vehicle = Bike
 In[9]: Motorbike 25
Out[9]: - : vehicle = Motorbike 25
In[10]: Car true
Out[10]: - : vehicle = Car true
```

Declaring functions on vehicles

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

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

```
 In[8]: Bike
Out[8]: - : vehicle = Bike
 In[9]: Motorbike 25
Out[9]: - : vehicle = Motorbike 25
In[10]: Car true
Out[10]: - : vehicle = Car true
```

Declaring functions on vehicles

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

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

```
  In[8]: Bike
 Out[8]: - : vehicle = Bike
  In[9]: Motorbike 25
 Out[9]: - : vehicle = Motorbike 25
 In[10]: Car true
Out[10]: - : vehicle = Car true
```

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

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

```
 In[8]: Bike
Out[8]: - : vehicle = Bike
 In[9]: Motorbike 25
Out[9]: - : vehicle = Motorbike 25
In[10]: Car true
Out[10]: - : vehicle = Car true
```

Declaring functions on vehicles

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

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

```
 In[8]:  Bike
Out[8]:  - : vehicle = Bike
 In[9]:  Motorbike 25
Out[9]:  - : vehicle = Motorbike 25
In[10]:  Car true
Out[10]: - : vehicle = Car true
```

Declaring functions on vehicles

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

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

```
In[8]: Bike
Out[8]: - : vehicle = Bike
In[9]: Motorbike 25
Out[9]: - : vehicle = Motorbike 25
In[10]: Car true
Out[10]: - : vehicle = Car true
```

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

Even though the constructors have different data, they are all of type vehicle when wrapped by the constructor.

```
In[11]: [ Bike; Car true; Motorbike 450 ]

Out[11]: - : vehicle list
```

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

Even though the constructors have different data, they are all of type vehicle when wrapped by the constructor.

```
In[11]: [ Bike; Car true; Motorbike 450 ]

Out[11]: - : vehicle list
```

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

Even though the constructors have different data, they are all of type vehicle when wrapped by the constructor.

```
 In[11]: [ Bike; Car true; Motorbike 450 ]
Out[11]: - : vehicle list
```

A finer wheel computation

```
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.