

Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 2 - Sequence 0: User-defined types



Typed functional programming

The next 2 weeks:

How to structure code and data with **types**?

Overview of Week 2

1. User-defined types
2. Tuples
3. Records
4. Arrays
5. Case study: A small typed database

Types as documentation

- ▶ A value of a primitive type can be used to **encode** some specific data.
- ▶ Example: $\text{day} = \{0, 1, 2, 3, 4, 5, 6\} \subset \text{int}$
- ▶ A type identifier carries an **informal invariant**.
- ▶ Example: An integer x is a valid day if $0 \leq x \leq 6$.
- ▶ Writing

`is_week_end : day -> bool`

informally means that integers between 0 and 6
are the only valid inputs for this function.

Representing colors using integers I

```
type color = int;;  
# type color = int  
let red    : color = 0;;  
# val red  : color = 0  
let white  : color = 1;;  
# val white : color = 1  
let blue   : color = 2;;  
# val blue  : color = 2
```

Type annotations I

```
type positive = int;;  
# type positive = int  
let abs (x : int) = (if x < 0 then -x else x : positive);;  
# val abs : int -> positive = <fun>  
let abs' (x : int) : positive = if x < 0 then -x else x;;  
# val abs' : int -> positive = <fun>
```

Syntax to declare a type

- ▶ To declare a type:

```
type some_type_identifier = some_type
```

- ▶ `some_type_identifier` is a **synonym** or **abbreviation** for `some_type`.
- ▶ A type identifier must start with a lowercase letter.
- ▶ For now, `some_type` can be `int`, `bool`, `string`, `char`, `float`.
- ▶ We will discover other type constructions soon!

Syntax to annotate with a type

- ▶ To annotate an identifier with a type:

```
let x : some_type = some_expression
```

- ▶ To annotate a function argument with a type:

```
let f (x : some_type) = some_expression
```

- ▶ To constrain the return type of a function:

```
let f x : some_type = some_expression
```

- ▶ To constrain the type of an expression:

```
let f x = (some_expression : some_type)
```

In the machine

- ▶ Type annotations have no impact on the program execution.
- ▶ Let “**type** `t = int`” and `x` be a value of type `t`, then `x` is also of type `int`.
- ▶ Hence, a value of type `t` is represented as a value of type `int` in the machine.

Pitfalls: Multiple type definitions

- ▶ In the REPL, be careful with unintended hiding of type identifiers.
- ▶ The error messages may be hard to understand.

Representing positive integers I

```
type t = int;;  
# type t = int  
let x : t = 0;;  
# val x : t = 0  
type t = bool;;  
# type t = bool  
let f (x : t) = not x;;  
# val f : t -> bool = <fun>  
let z = f x;;  
# Characters 10-11:  
  let z = f x;;  
      ^  
Error: This expression has type t/1016 = int  
      but an expression was expected of type t/1018 = bool
```

Pitfalls: Limitations of type synonyms

- ▶ Consider `type positive = int`.
- ▶ The type synonym `positive` is only a documentation.
- ▶ It does not provide **more static guarantees about positivity** than `int`.
- ▶ For instance, the following code is accepted by the type-checker:

```
let x : positive = -1
```

- ▶ *OCaml* provides many ways to define **more precise types**.