Functions

CS3100 Fall 2019

Review

Previously on CS3100

- Syntax and Semantics
- Expressions: if, let
- Definitions: let

Today

- Functions

Anonymous Function

OCaml has support for anonymous function expressions. The syntax is

```
fun x1 ... xn -> e
```

- A function is a value; no further computation to do.
- In particular, `e` is not evaluated until the function is applied.

Anonymous Functions

In [1]:

```
fun x -> x + 1
```

Out[1]:

```
- : int -> int = <fun>
```

The function type `int -> int` says that it takes one argument of type `int` and returns a value of type `int` .

Thunk

In [2]:

```
fun () -> 1
```

Out[2]:

```
- : unit -> int = <fun>
```

- A function of type `unit -> t` is called a `thunk`.
- Delay the computation of the RHS expression until application.

Anonymous Functions

The function body can refer to any variables in scope.

In [3]:

```
let foo =
  let y = 10 in
  let x = 5 in
  fun z -> x + y + z
```

Out[3]:

```
val foo : int -> int = <fun>
```

Functions are values

Can use them *anywhere* we can use values:

- Functions can **take** functions as arguments
- Functions can **return** functions as arguments

As you will see, this is an incredibly powerful language feature.

Function application

The syntax is

```
e0 e1 ... en
```

- No parentheses necessary

Function Application Evaluation

```
e0 e1 ... en
```

- Evaluate `e0 ... en` to values `v0 ... vn`
- Type checking will ensure that `v0` is a function `fun x1 ... xn -> e`
- Substitute `vi` for `xi` in `e` yielding new expression `e'`
- Evaluate `e'` to a value `v`, which is result

Function Application

In [4]:

```
(fun x -> x + 1) 1
```

Out[4]:

```
- : int = 2
```

In [5]:

```
(fun x y z -> x + y + z) 1 2 3
```

Out[5]:

```
- : int = 6
```

The above function is syntactic sugar for

In [6]:

```
(fun x -> fun y -> fun z -> x + y + z) 1 2 3
```

Out[6]:

```
- : int = 6
```

Multi-argument functions do not exist!

Function definition

We can name functions using `let`.

```
let succ = fun x -> x + 1
```

which is semantically equivalent to

```
let succ x = x + 1
```

You'll see the latter form more often.

Function definition

In [7]:

```
let succ x = x + 1
```

Out[7]:

```
val succ : int -> int = <fun>
```

In [8]:

```
succ 10
```

Out[8]:

```
- : int = 11
```

Function definition

In [9]:

```
let add x y = x + y
```

Out[9]:

```
val add : int -> int -> int = <fun>
```

In [10]:

```
let add = fun x -> fun y -> x + y
```

Out[10]:

```
val add : int -> int -> int = <fun>
```

In [11]:

```
add 5 10
```

Out[11]:

```
- : int = 15
```

Partial Application

```
(fun x y z -> x + y + z) 1
```

returns a function

```
(fun y z -> 1 + y + z)
```

In [12]:

```
let foo = (fun x y z -> x + y + z) 1
```

Out[12]:

```
val foo : int -> int -> int = <fun>
```

In [13]:

```
foo 2 3
```

Out[13]:

```
- : int = 6
```

Partial Application

A more useful partial application example is defining `succ` and `pred` functions from `add` .

In [14]:

```
let succ = add 1
let pred = add (-1)
```

Out[14]:

```
val succ : int -> int = <fun>
```

Out[14]:

```
val pred : int -> int = <fun>
```

In [15]:

```
succ 10
```

Out[15]:

```
- : int = 11
```

In [16]:

```
pred 10
```

Out[16]:

```
- : int = 9
```

Recursive Functions

Recursive functions can call themselves. The syntax for recursive function definition is:

```
let rec foo x = ...
```

Notice the `rec` key word.

Recursive Functions

In [17]:

```
let rec sum_of_first_n n =
  if n <= 0 then 0
  else n + sum_of_first_n (n-1)
```

Out[17]:

```
val sum_of_first_n : int -> int = <fun>
```

In [18]:

```
sum_of_first_n 5
```

Out[18]:

```
- : int = 15
```

Tracing functions in Jupyter

Jupyter (really the ocaml top-level behind the scenes) provides support for tracing the execution of functions.

In [19]:

```
#trace sum_of_first_n;;

sum_of_first_n 3;;
```

```
sum_of_first_n is now traced.
sum_of_first_n <-- 3
sum_of_first_n <-- 2
sum_of_first_n <-- 1
sum_of_first_n <-- 0
sum_of_first_n --> 0
sum_of_first_n --> 1
```

Out[19]:

```
- : int = 6
```

In [20]:

```
#untrace sum_of_first_n
```

```
sum_of_first_n --> 3
sum_of_first_n --> 6
```

Exercise

Implement a recursive function that computes the nth fibonacci number. The fibonacci sequence is `[0;1;1;2;3;5;8;...]`.

In [21]:

```
let rec fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n-2) + fib (n-1)
```

```
sum_of_first_n is no longer traced.
```

Out[21]:

```
val fib : int -> int = <fun>
```

In [22]:

```
assert (fib 10 = 55)
```

Out[22]:

```
- : unit = ()
```

Mutually recursive functions

In [23]:

```
let rec even n =
  if n = 0 then true
  else odd (n-1)

and odd n =
  if n = 0 then false
  else even (n-1)
```

Out[23]:

```
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
```

In [24]:

```
odd 44
```

Out[24]:

```
- : bool = false
```

Recursing too deep

Let's invoke `sum_of_first_n` with larger numbers.

In [25]:

```
sum_of_first_n 1
```

Out[25]:

```
- : int = 1
```

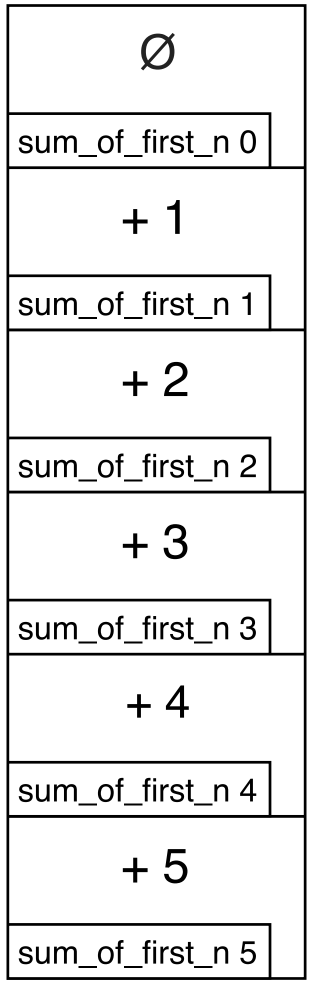
Stack buildup

```
let rec sum_of_first_n n =
  if n <= 0 then 0
  else n + sum_of_first_n (n-1)
```

Some work `"+ n"` left to do after the recursive call returns. This builds up stack frames.

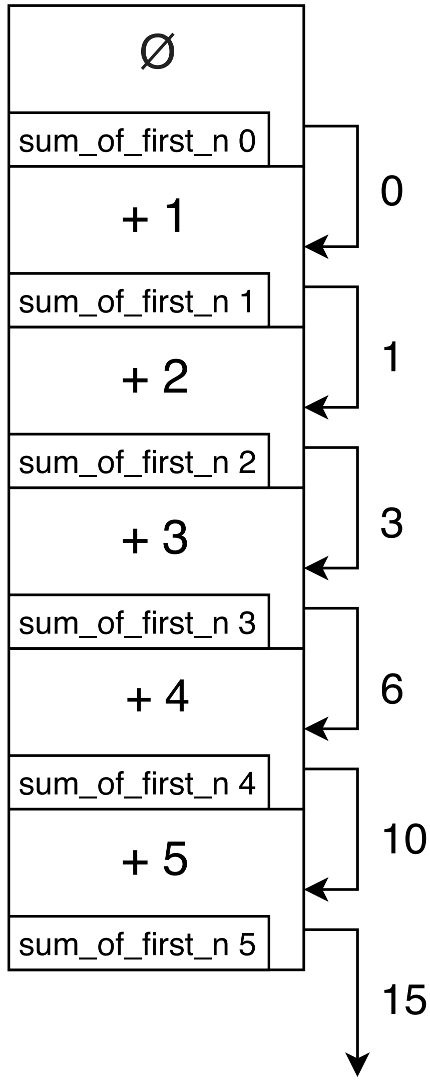
Stack buildup

For `sum_of_first_n 5` :



Stack buildup

For `sum_of_first_n 5` :



Tail recursion

Rewrite the function such that the recursive call is the last thing that the function does:

In [26]:

```
let rec sum_of_first_n_tailrec n acc =
  if n <= 0 then acc
  else sum_of_first_n_tailrec (n-1) (n + acc)
```

Out[26]:

```
val sum_of_first_n_tailrec : int -> int -> int = <fun>
```

In [27]:

```
sum_of_first_n_tailrec 10000 0
```

Out[27]:

```
- : int = 50005000
```

Tail recursion

```
let rec sum_of_first_n_tailrec n acc =
  if n <= 0 then acc
  else sum_of_first_n_tailrec (n-1) (n + acc)
```

- No work left to do when the recursive call returns except return result to caller.
- OCaml compiler does **tail call optimisation** that pops current call frame before invoking recursive call.
  - No stack buildup => equivalent to writing a tight loop.

Fin.