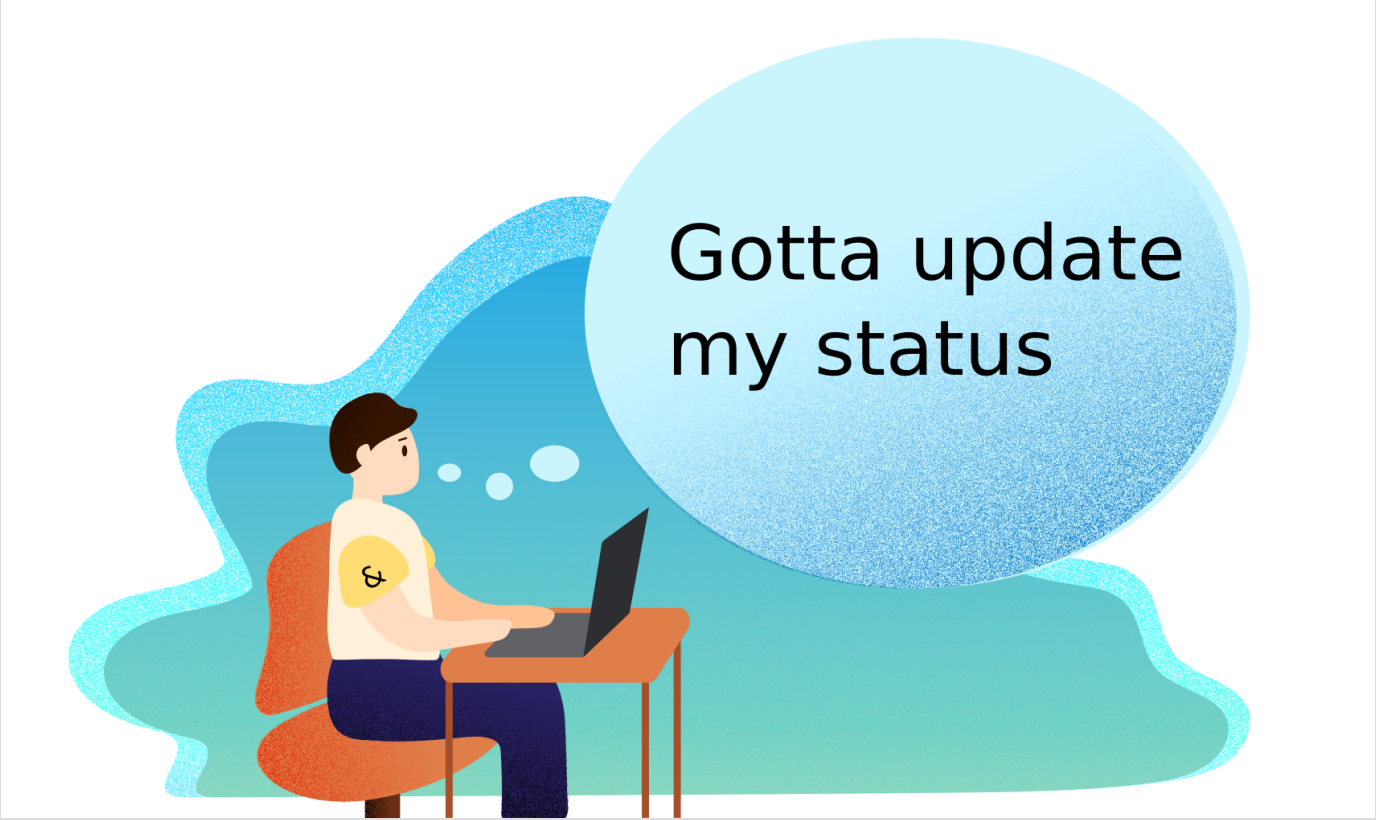
Imperative Programming

Having an effect

15-150 M21

Lecture 0802
02 August 2021

0 Effects in SML



Effect: A change to the state of the computer or the world

Value: A piece of data which is "fully calculated" or "fully simplified" – the kind of thing that can serve as an answer to a computational question (need to specify what this means)

Purity

Recall we had the notion of **purity**:

- An expression is pure if evaluating it causes no side-effects
- A function is pure if applications of it never cause side-effects
- A language is pure if it doesn't have any kind of effects

So far, we've been working with just the pure fragment of SML. But SML is not completely pure...

Demonstration:

`print`

```
print : string -> unit
```

We're generally not interested in pure `unit`-returning functions: for each type `t`, there's only one pure, total function of type `t -> unit`.

Usually, functions which return a `unit` are *impure*: we're executing them for their effect.

Demonstration:

`before, ignore, and ;`

1 Refs

SML has a built-in datatype called `ref`

```
datatype 'a ref = ref of 'a
```

But `ref`s are special: the data inside a `ref` cell is **mutable**, using the reassignment operator, `:=`.

Ref methods

```
(op :=) :  'a ref * 'a -> unit

!  :  'a ref -> 'a
```

Demonstration:
The `Ref` structure

Now with fewer bugs!

5 minute break

Warning #1:
Keep it fresh

A simple example

```
val r1 = ref 0
val r2 = r1
val r3 = ref 0
```

`r2` is just an *alias* for `r1`, whereas `r3` is an independent ref cell.

Key Point:

One ref cell is created for every use of the `ref` constructor

Warning #2:
Stage carefully

0802.0 (refs.sml)

```
2  fun f1 x y =
3  let
4    val r = ref x
5  in
6    if y<(!r)
7    then
8  !r + y before r:=y
9    else !r + y
10 end
```

0802.1 (refs.sml)

```
13 fun f2 x =
14 let
15   val r = ref x
16 in
17   fn y =>
18     if y<(!r)
19     then
20 !r + y before r:=y
21     else !r + y
22 end
```

When `ref`s and effects are involved, we find ourselves in an annoying situation: evaluating the same code at different times can give different results.

Our old definition of extensional equivalence is inadequate to guarantee **referential transparency**: if `e1` and `e2` are impure, then `e1` $\cong$ `e2` does not mean `e1` and `e2` are interchangeable, because they might have different side effects.

2 Other Effects

Colors!


```
(* YY, with foreground color 0<=XX<=255 *)
    "\^[[38;5;XXmYY\^[[0m"
(* YY, with background color 0<=XX<=255 *)
    "\^[[48;5;XXmYY\^[[0m"
```

Demonstration:

`OS.Process.system`

Demonstration:

File I/O

Module:

`Timing`

Summary

- SML is not a pure functional language: there are ways to cause effects
- We can use ref cells to have mutable data
- Effects introduce extra headache when reasoning about code

Next Time

Review!

Thank you!