Lambdas

The central concept in functional programming

15-150 M21

Lecture 0526
26 May 2021

0 To evaluate, or not to evaluate?

Demonstration: Declaration tracing

Lab Question

```
if b then e1 else e2
```

```
let
  val v1 = e1
  val v2 = e2
in
  if b then v1 else v2
end
```

Key Fact:

SML is an **eager** or **call-by value** language: the arguments of a function are evaluated all the way to values *before* being substituted into the body of the function

E.g. consider a function `f : int -> int` such that

$$\texttt{f(x)} \implies \texttt{if true then 5 else x}$$

What happens when you evaluate `f(3 div 0)`?

Eagerness of “multivariable” functions

Recall that evaluation of tuples is left-to-right, so to evaluate `f(e1,e2)`, we

- First evaluate `e1` to a value `v1`
- Then evaluate `e2` to a value `v2`
- Then compute `f(v1,v2)`

Recall that SML allows us to "infix" functions of 2 variables, and that the `op` keyword un-infixed them, so we could check their type.

```
(op +)   : int * int -> int
(op * )  : int * int -> int
(op -)   : int * int -> int
(op div) : int * int -> int
(op >)   : int * int -> bool
(op =)   : int * int -> bool
(op =)   : string * string -> bool
```

op orelse?

Shortcircuiting of `orelse`

```
true orelse ((2 mod 0)=1)
```

We don't need to evaluate `((2 mod 0)=1)` here, since the first expression is already true.

- If `e1` $\hookrightarrow$ `true`, then `(e1 orelse e2)` $\hookrightarrow$ `true` without ever evaluating `e2`
- If `e1` $\hookrightarrow$ `false`, then `(e1 andalso e2)` $\hookrightarrow$ `false` without ever evaluating `e2`

This means that `orelse` and `andalso` are *not* functions (contradicts eagerness)!

1 Declaring and Applying Functions

What are the values of type `T1 -> T2`?

The syntax for an expression of type `t1 -> t2` is

```
fn x => e
```

where `e : t2` (possibly using the variable `x:t1`)

0526.1 (fn.sml)

```
val half : int -> int = fn x => x div 2
```

0526.2 (fn.sml)

```
val isNegative : int -> bool = fn x => x<0
```

0526.3 (fn.sml)

```
val apTen : (int -> int) -> int = fn f => f(10)
```

Demonstration: Evaluating with lambdas

Course slogan:

Functions are values

- Functions are pieces of data which can be passed around:

0526.4 (fn.sml)

```
val |> : int * (int -> string) -> string =
    fn (x,f) => f x
infix |>
val "2" = 2 |> Int.toString
```

- Lambda expressions are values

`fn x=>2+2` does not evaluate to `fn x => 4`

`fn x => 1 div 0` is a value

Not everything of an arrow type is a value

```
let
  val k = 1 div 0
in
  fn x => x
end
```

What should this do?

0526.5 (closure.sml)

```
val foo : int = 4 + 5
val bar : int -> int =
  fn x => foo div (foo - x)
val foo : int = 6
val y : int = bar foo
```

Whenever a function value is declared, SML stores *two* pieces of information as part of the binding:

- The `fn` value
- A "snapshot" of all the bindings in the environment at the time. This snapshot is called the **closure** of the function

Whenever that function is called, SML will use the *closure* to substitute variables in the function body.

Demonstration: Declaration tracing with closures

What if we want to implement *recursive* functions? For instance, the exponential function can be implemented recursively using the following mathematical fact:

$$2^0 = 1$$
$$2^n = 2 \cdot 2^{n-1} \qquad (n > 0)$$

0526.6 (fun.sml)

```
val exp : int -> int =
  fn n => if n=0 then 1 else 2*exp(n-1)
```

SML provides the fun keyword to declare a function value which is allowed to refer to itself

0526.7 (fun.sml)

```
fun exp (n:int):int =
  if n=0 then 1 else 2 * exp(n-1)
```

5-minute break

2 Documenting Functions

Words of wisdom:

Programs must be written for people to read, and only incidentally for machines to execute

A primary purpose of types is as *documentation*: the type of a function tells you a lot of information about what that function is.

When providing documentation of your code, at a minimum you must say what types each of the functions have

0526.7 (fun.sml)

```
(*  exp : int -> int
```

Specifying that `exp : int -> int` begins to document it, but we would also want to tell a user not to apply `exp` to a negative number.

A **precondition** is a logical statement constraining what inputs are allowed to a function.

0526.7 (fun.sml)

```
(*  exp : int -> int
 *  REQUIRES: n>=0
```

By convention, we write `REQUIRES: true` to mean that there is *no* precondition – any input of the correct type suffices.

Next, we want to tell our user what the function will *do* when given an input that satisfies the REQUIRES. We call this a **postcondition**.

0526.7 (fun.sml)

```
(*  exp : int -> int
 *  REQUIRES: n>=0
 *  ENSURES: exp(n) == 2^n
```

The type, precondition, and postcondition form the **specification** of a function. `f` is said to **satisfy** its spec if `f` has the appropriate type, and for every `v` of the input type satisfying the REQUIRES, `f(v)` satisfies the ENSURES.

As (probably) covered in lab, you can test your function by writing `val` declarations where the right-hand side is a value you're not allowed to shadow.

0526.7 (fun.sml)

```

val 1 = exp 0
val 131072 = exp 17
```

Be sure you're actually performing the test, and not actually shadowing something!

The 5-step methodology

Whenever you implement any function you should:

1. Specify the type
2. Write an appropriate REQUIRES (weak as possible)
3. Write an appropriate ENSURES (strong as possible)
4. Implement the function
5. Write enough test cases

0526.7 (fun.sml)

```
(*  exp : int -> int
 *  REQUIRES: n>=0
 *  ENSURES: exp(n) == 2^n
 *)
fun exp (n:int):int =
  if n=0 then 1 else 2 * exp(n-1)

val 1 = exp 0
val 131072 = exp 17
```

Totality

Defn. A function value `f : t1 -> t2` is said to be **total** if, for all values `v : t1`, the expression `f(v)` is *valuable*.

Examples:

- `(fn s => s)`
- `op +`
- `Int.toString`

Non-examples:

- `div`
- `exp`

When are function expressions extensionally equivalent?

Recall referential transparency: extensionally-equivalent expressions are interchangeable in code. So if `f` $\cong$ `g`, then we need `f` and `g` to behave exactly the same.

Defn. Two expressions `f`, `g` of type `t1 -> t2` are **extensionally equivalent** if for all values `v : t1`,

$$\texttt{f(v)} \cong \texttt{g(v)}$$

3 Patterns

exp traces are clunky

```
exp 4
⟹ if 4=0 then 1 else 2*exp(4-1)
⟹ 2*exp(3)
⟹ 2*(if 3=0 then 1 else 2*exp(3-1))
⟹ 2*(2*exp(2))
⟹ 2*(2*(if 2=0then 1 else 2*exp(2-1)))


⟹ 2*(2*(2*exp(1)))
⟹ 2*(2*(2*(if 1=0 then 1 else 2*exp
 (1-1))))
                    )))
```

There's a better way...

0526.8 (patterns.sml)

```
fun exp (0:int):int = 1
  | exp n             = 2 * exp(n-1)
```

Much better...

```
exp 4
⟹ 2 * exp(3)
⟹ 2 * 2 * exp(2)
⟹ 2 * 2 * 2 * exp(1)
⟹ 2 * 2 * 2 * 2 * exp(0)
⟹ 2 * 2 * 2 * 2 * 1
⟹ 16
```

Pattern Matching

In this example, `0` and `n` are **patterns** that SML is **matching** against.

When pattern matching, SML will try to match with each of the patterns in the order they're written, and step into the first clause it matches with.

0526.9 (patterns.sml)

```
fun zeros (0:int,0:int):string = "Both"
  | zeros (0,n) = "First"
  | zeros (m,0) = "Second"
  | zeros (m,n) = "Neither"
```

0526.10 (patterns.sml)

```
fun zeros' (0:int,0:int):string = "Both"
  | zeros' (0,_) = "First"
  | zeros' (_,0) = "Second"
  | zeros' _ = "Neither"
```

0526.11 (patterns.sml)

```
fun zeros'' (n:int,m:int):string =
  case (n,m) of
       (0,0) => "Both"
     | (0,_) => "First"
     | (_,0) => "Second"
     | _ => "Neither"
```

- Lambda expression clauses:

```
val isZeroOrOne : int -> bool
    = fn 0 => true | 1 => true | _ => false
```

- `val` declarations

```
val 8 = exp 3
```

Allowed patterns

- Constructors

```
fn true => e1 | false => e2
```

- Variable names

```
fn (x:int) => x
```

- Wildcards

```
fn (_ : string) => 2
```

- Tuples of patterns

```
fun foo ((0,0),_) = "a"
  | foo ((_,0),(7,_)) = "b"
  | foo ( _, (8,8)) = "c"
  | foo _ = "d"
```

Not patterns

- Function applications

```
(* Doesn't work *)
val m+n = 2
val (s1 ^ s2) = "hello world"
```

- Non-match-able types

```
(* Doesn't work *)
val (fn x => e) : int -> string = f
```

- Repetitive patterns

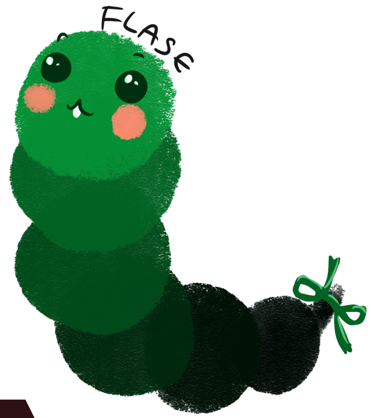
```
(* Doesn't work *)
fun equal (m:int,m:int) = true
  | equal _ = false
```

Note: the following are equivalent:

```
case b of
  true => e1
| false => e2
```

```
if b then e1 else e2
```

Common error: the “flase” bug




```
case b of
  flase => 2
| true => 1
```

Proving the valuability of exp

Prop. For all values `n : int` with $n \geq 0$, `exp(n)` is valuable.

Proof. by induction on `n`.

BC: `n=0`.

`exp 0` $\Longrightarrow$ `1.` (first clause, `exp`)

IH: Suppose for some `n>=0`, `exp(n)` is valuable.

WTS: `exp(n+1)` is valuable.

`exp(n+1)` $\Longrightarrow$ `2 * exp(n)` (second clause, `exp`)

$\Longrightarrow$ `2 * v` (for some value `v`, by IH)

$\Longrightarrow$ `v'` (for some value `v'`, by totality of `op*`)

Summary

- SML provides ways to control when expressions get evaluated
- Shadowing is not reassignment: the old binding is remembered, particularly in function closures
- Functions are specified by their applicative behavior
- Pattern matching facilitates concise, elegant function declarations

Next Time

- Recursion & Induction
- Strong Induction
- Recurrences & sequential runtime analysis

Thank you!