Bindings and Functions

COMP 302

PART 02

Binding

An ML program consists of a sequence of **bindings** or declarations

Example of simple program

```
val x : int = 17;

val y : int = 121;

val z : int = (x + y) + (y - 100);

val q : int = z * 3;

val my_abs :int = if (z<0) then 0-z else z;
```

Binding Values

We begin by looking at bindings of values. Later we consider other forms of binding (datatypes).

Bind value to variable (syntax): `val  <var>  :  <type>  = <exp>`

- e.g. val x : int = 2+3

A binding is well-typed if <exp> : <type>.

The variable <var> can be used in the scope of the declaration (to be defined later) as an expression of type <type>

Evaluation: To evaluate a sequence of val declarations, evaluate the first expression and then substitute its value for the variable in subsequent declarations

Variables are identifiers that name values (not memory cells)

When a binding is established, the variable names the same value until it goes out of scope

SML variables are **immutable**

Evaluating a Sequence of Bindings

```
val x : int = 2+3

val y : int = x + 1

val z : int = x + y
```

first evaluate 2+3 then substitute 5 for x:

```
val x : int = 5

val y : int = 5 + 1

val z : int = 5 + y
```

Evaluating a Sequence of Bindings

then evaluate 5+1 and substitute for y

```
val x : int = 5
val y : int = 6
val z : int = 5 + 6
```

then

```
val x : int = 5
val y : int = 6
val z : int = 11
```

Shadowing

```
val x : int = 2+3
val y : int = x + 1
val x : int = 7
val z : int = x + 1
```

There are two different variables that happen to have the same name. Unlike C, Java etc., the value of a variable doesn’t change once it has been bound. They are like variables in math.

The x in line 2 is an occurrence of the variable bound to the value 5 in line 1. The x in line 4 is an occurrence of the one bound to 7 in line 3.

Shadowing

The semantics of these bindings can be clarified by consistently renaming one of the variables, say the second, to x’

```
val x : int = 2+3
val y : int = x + 1
val x’ : int = 3
val z : int = x’ + 1
```

Bindings are **immutable**

What’s missing so far?

There are features of imperative languages missing in SML (at least so far)

- Mutability (no assignment)
- Statements: everything is an expression

Practicalities

Note: A sequence of bindings can be saved in a file with suffix .sml

We can add a sequence of bindings from a file with **use “test.sml”**

Result of “use” command is () of type unit but it also includes all of the bindings in the file test.sml

Function Declarations

ML functions are similar to mathematical functions.

Definition:

For example the mathematical function f(x) = 2x+y can be defined as

```
fun f (x : real) : real = (2.0 * x) + 6.0
```

We define a function named f.

It has an argument x of type real and produces a result of type real,

The body is (2.0 * x) + 6.0

Function Application

Function application is the main operation done with functions

Function application calculates by substitution. Plug the value of the argument for the variable in the body:

```
f (3.0) |-> (2.0 * 3.0) + 6.0
        |-> 6.0 + 6.0
        |-> 12.0
```

In ML the argument is calculated until it yields a value before the body is evaluated

```
f (4.0 – 1.0) |-> f (3.0)
        |-> …
```

This semantics is known as eager evaluation or “call-by-value”

Function Application

An alternative used by some languages is lazy evaluation or “call-by-name”

```
f (4.0 – 1.0) |-> (2.0 * (4.0 – 1.0)) + 6.0
      |-> (2.0 * 3.0) + 6.0
      |-> …
```

The obvious difference can be seen if the argument is 5 div 0 and the variable is not used in the application

Later we see how to simulate this in ML

Call-by-value makes it easier to predict when an expression is evaluated which helps in analyzing the time complexity and helps for reasoning about certain expressions

Scoping

```
val pi : real = 3.14;
fun area (x:real) : real = pi * x * x;
val a : real = area (2.0);   (* value of a is 12.56 *)
val pi : real = 3.0;         (* this pi shadows the first *)

val b : real = area (2.0);
```

(Aside: note the use of (* … *) for comments)

What is the value of b?

Static Scoping

```
val pi : real = 3.14;
fun area (x:real) : real = pi * x * x;
val a : real = area (2.0);  (* value of a is 12.56 *)
val pi : real = 5.0;        (* this pi shadows the first *)
val b : real = area (2.0);
```

What is the value of b?

Using static scope rules(* value of b is 12.56 *)

With static scope rules, the body of a function is evaluated in the dynamic environment that exits at the point of definition

SML uses static scoping

Dynamic Scoping

```
val pi : real = 3.14;
fun area (x:real) : real = pi * x * x;
val a : real = area (2.0);  (* value of a is 12.56 *)
val pi : real = 3.0;        (* this pi shadows the first *)
val b : real = area (2.0);
```

What is the value of b?

Using dynamic scope rules (* value of b is 12.0 *)

With dynamic scope rules the body of the function is evaluated in the environment that exists at the point of usage

Dynamic scope rules were used in the past in early versions of Lisp and some other languages. It is used in some languages (e.g. Java) for exception handlers

Dynamic scope makes the evaluation of function application very simple and efficient. However it is more difficult to reason about and understand programs

Function Types

```
fun f (x : real) : real = (2.0 * x) + 6.0
```

The type of f is `real -> real`

Functions are not some special creatures but just regular old values that have a type like everything else in ML

We can have functions of type `<t1> -> <t2>`

- Values of this type are all functions defined by function bindings where the argument if of type <t1> and the body is of type <t2>
- Operations on this type include function application: f a. If f has type <t1> -> <t2> and a has type <t1> then f a has type <t2>

Consequences: Functions can be passed as arguments to other functions and returned from functions as results.

Semantics of Functions

We will review what we have been discussing about function definition and application a bit more formally.

For now we focus on functions of a single argument.

Later we introduce tuples to handle functions of several arguments and still later we discuss Currying where all functions can be reduced to single arguments

Function Definitions

Syntax: `fun foo (x : t1) : t2 = e`

Evaluation: A function is a value whose name is added to the environment

Type-checking
- Check that e has type t2 in an environment with
  - Previous static bindings
  - x : t1
  - foo : t1 -> t2 (to allow for recursion)
- Then add the binding name : t1 -> t2

Return type is t2, the type of e (the body)

The type of value t2 is often dropped but then we rely on type inference to determine it

The type of “foo” is set at the point of definition and not in earlier bindings

The argument x can only be used in e (i.e. its scope it limited to the body)

Function Application

syntax: `e0 (e1)` or `e0 e1`

type-checking:

- e0 must have type t1 -> t2
- e1 must have type t1
- the expression has type t2

evaluation rules:

- evaluate e0 to v0 (which must be a function type)
- evaluate e1 to v1, using dynamic environment at the point of call (this is call by value)
- extend the dynamic environment to map x to v1
- which dynamic environment – the environment at the point of definition, not at the point of call (e0 included for recursion) (this is the static scoping rule we define)
- evaluate the function body e in this environment

Recursive Functions

Of course functions can be recursive

In the next lecture we will look at recursive functions derived from inductively specified data

For now we just look at a simple example

This is a function on two variables be

Later we will see that this is just a shorthand for a function that takes one variable which is a pair of values

A Recursive Function Example

```
fun pow (x:int, y:int) =       (* correct only for y >= 0 *)
    if y=0
    then 1
    else x * pow(x,y-1)

fun cube (x:int) =
    pow(x,3)

val sixtyfour = cube(4)

val fortytwo = pow(2,4) + pow(4,2) + cube(2) + 2
```

Evaluation by Substitution

Evaluating a recursive function by substitution:

```
pow (6,2)
|-> if (2=0) then 1 else 6*pow(6,2-1)
|-> if (2=0) then 1 else 6*pow(6,1)
|-> 6 * pow(6,1)
|-> 6 * if (1=0) then 1 else 6*pow(6,1-1)
|-> 6 * if (1=0) then 1 else 6*pow(6,0)
|-> 6 * 6 * pow(6,0)
|-> 6 * 6 * if (0=0) then 1 else 6*pow(6,0-1)
|-> 6 * 6 * 1
|-> 36
```

What Else is Missing

There are no loops in SML

Repetition is accomplished by recursion