Local Definitions, Lists

COMP 302

PART 04

Local Definitions

A let expression has the form:

```
let b1 b2 … bn in e end
where each bi is a binding and e is an expression
```

Type-checking

- Check each bi in order and then e in a static environment that includes all previous bindings
- The type of the expression is the type of e

Evaluation:

- Evaluate each bi in sequence and e in a dynamic environment that includes all previous bindings
- The result is the result of evaluating e

Note: The scope of a binding consists of the later bindings and the body, e

Examples of Let Expressions

```
let val x : int = 2 * 7
    val y : int = x - 4
in x + y
end
The value of this expression is 24
```

Note: After 14 is bound to x, it is substituted for subsequent occurrences of x and so the multiplication is done only once

Nested Lets

```
let val x = 1
in
 (let val x = 2 in x+1 end)+(let val y = x+2 in y+1 end)
end
```

The value of the expression is 7

The second binding of x shadows the first in the nested let expression

Local Functions

"Helper" functions are often used to compute other functions

They have little use otherwise

They can be defined locally

Binding Functions Locally

```
fun isPrime (n : int) : bool =
  let fun noMoreDivisors (m : int) : bool =
        if n mod m = 0 then false
         else if m*m >= n then true
         else noMoreDivisors (m+1)
   in
     noMoreDivisors (2)
   end
```

Newton's Root Finding

```
fun sqRoot (x : real) : real =
  let
    val epsilon = 0.0001
    fun approxRoot (guess : real) : bool =
      abs (guess * guess - x) < epsilon
    fun improve (guess : real) : real =
      (guess + x / guess) / 2.0
    fun iterate (guess : real) : real =
      if approxRoot(guess) then guess
                           else iterate(improve(guess))
  in
    iterate (1.0)
  end
```

Lists

A list of integers (of type int list) is the smallest set containing either

- [] (pronounced “nil” or “the empty list”, or
- x :: xs where x : int and xs : int list (where :: is pronounced cons)

The values are of the form 1 :: (2 :: (3 :: (4 :: []))) which can be written 1 :: 2 :: 3 :: 4 :: [] because cons (:: is right associative)

and we abbreviate as [1, 2, 3, 4] in SML

This is a structurally recursive definition and functions over lists are generally structurally recursive following this pattern

Deconstructing Lists

SML, like other functional languages such as Lisp, Scheme has functions that allow access to list values

There are three functions:

- null evaluates to true for empty lists and false for nonempty lists
- hd returns the head, the first element of a list and raises an exception if the list is empty
- tl returns the tail of the list, everything except the head, and raises an exception if the list is empty

Example:

```
Fun sum_list (lst : int list) : int =
  if null lst
  then 0
  else hd (lst) + sum_list(tl(lst))
```

Pattern Matching on Lists

Although functions such as hd and tl are common in some languages, they tend to be error prone and might be applied to empty lists, raising exceptions

In SML pattern matching is usually used for processing lists

For example, to compute the length of a list of strings (note that we can have lists of any type)

```
fun length (lst : string list) : int =
  case lst of
      [] => 0
    | h::t => 1 + length (t)
```

Length with Pattern Matching

```
fun length (lst : int list) : int =
  case lst of
     [] => 0
   | x :: xs => 1 + length xs
Evaluation by substitution”
length (1 :: (2 :: []))
|-> case (1 :: (2 :: [])) of [] => 0| x :: xs => 1 + length xs
|-> 1 + length (2 :: [])
… |-> 2
```

Sum with Pattern Matching

```
fun sum (lst : int list) : int =
  case lst of
    [] => 0
  | x :: xs => 1 + sum xs
```

```
Evaluation also proceeds in a similar way
```

Note the similarity in patterns.

Increment values in lst by amt

```
fun incr (lst : int list, amt : int) : int list =
  case lst of
    [] => []
  | x :: xs => (x + amt) :: incr(xs, amt)
```

This function takes two arguments – really a single argument which is a pair – and evaluates to a list

```
incr ([1, 2, 5, 9], 500) = [501, 502, 505, 509]
```

Proving Properties by structural induction

We will demonstrate the method of proving the properties of a program using structural induction

Theorem 1: `incr (incr (lst, a) , b) == incr (lst, a+b)`

for all values of lst : int list, a : int, b : int

That is, two nested calls of incr can be fused into one call. Rather than traversing the list twice we can optimize and only traverse the list once

The proof is by structural induction on lst

Proof : base case

Base case: Show that `(incr (incr ([], a), b) == incr ([], a+b)`

```
(incr (incr ([], a), b)
 == incr (case [] of [] => [] | x::xs => (x + a)::incr(xs, a), b)
 == incr ([], b)
 == case [] of [] => [] | x::xs => (x + b)::incr(xs, b)
 == []
incr ([], a+b)
 == case [] of [] => [] | x::xs => (x + (a+b)::incr(xs,(a+b))
 == []
```

Inductive Step

Inductive hypothesis:

```
incr (incr (xs, a) , b) == incr (xs, a+b)
```

Show:

```
incr (incr (x::xs, a) , b) == incr (x::xs, a+b)
```

Start of Proof

```
incr (incr (x::xs, a) , b) == incr (x::xs, a+b)
== incr (case x::xs of [] => [] | x :: xs => (x+a)::incr(xs,a), b)
== incr ((x + a)::incr(xs, a), b)
?? case (x+a)::incr(xs, a) of [] => [] | x :: xs => (x+b)::incr(xs,b)
```

Can we justify this last step?

Previously we substituted an argument for a parameter where the argument was a value

Now we are trying to substitute an expression that has to be evaluated for a parameter.

When does this work?

What we want

Given a function:

fun f (x : A) : B = e1 and an expression e2, we want to be able to substitute e2 for all instances of x in e1 and end up with an equivalent expression

We can use the notation (f e2) == [e2/x]e1 for any e2,

Where [e2/x]e1 is the expression we get when we substitute e2 for x in e1

This is not always true. Substitution doesn't always work:

For example, for fun f (x : int) int = 7

Substitute the expression f(1 div 0) for all occurrences of x in e1 (7) and get f(1 div 0) == 7, which is clearly not the case in SML

This would imply that 7 == raise Div

Valuable

We have to be more careful about when substitution of expressions works.

We want to restrict ourselves to expressions which can be evaluated (ie have values)

Definition: expression e is valuable iff there is some value v such that e == v

Note that:

- If e = (e1,e2) then e is valuable iff e1 is valuable and e2 is valuable
- If e = e1 + e2 then e is valuable iff e1 is valuable and e2 is valuable
- If e = e1 :: e2 then e is valuable iff e1 is valuable and e2 is valuable

Definition: a function f : A -> B is total iff for all values v : A, (f v) is valuable

Note: if f is total and e1 is valuable then e = (f e1) is valuable

Substitution in Proof

Suppose we have a function defined as `fun f (x : A ) : B = e1`

If the expression e2 : A is valuable then (f e2) == [e2/x]e1

We can continue with the earlier proof because all the arguments to the operations are values and the operators +, ::, pairing, and incr are total (the last really requires proof)

Continuing Proof

```
incr (incr (x::xs, a) , b) == incr (x::xs, a+b)
== incr (case x::xs of [] => [] | x :: xs => (x+a)::incr(xs,a), b)
== incr ((x + a)::incr(xs, a), b)
== case (x+a)::incr(xs, a) of [] => [] | x :: xs => (x+b)::incr(xs,b)
```

We have just justified this last step and we can continue

```
== ((x+a)+b) :: incr (incr(xs,a),b)
== ((x+a)+b) :: incr(xs, a+b) , by Induction Hypothesis
== (x + (a+b)) :: incr (xs, a+b) , by mathematics
```

Completing Proof

To complete the proof, we look at the RHS

```
incr (x::xs, a+b)
== case x::xs of [] => []|x:xs => (x+(a+b))::incr(xs,(a+b))
== (x + (a+b)) :: incr (xs, a+b)
```

Note that this proof is not by mathematical induction on the length of the list. It is based on the structure of the list and does not mention the length at all

Another example

We claimed that incr was total

We outline a proof of this as another example of structural induction

Theorem 2: incr is total

Proof: We have to shot that for any value lst of type int list, and for any value a of type int, there exists a list v such that incr (lst, a) == v

Proof

Base case:

- We show that incr ([], a) == [], so the value v above can be taken to be []

Induction Hypothesis

- There is some value, v of type int list such that incr (xs, a) == v

Induction Step

- Expand incr (x::xs, a) to get (x+a)::incr(xs,a) and since + is total, :: is total and by the IH, the function is total