Control Structures

CS152
Chris Pollett
Nov. 24, 2008.

Outline

- More on Polymorphism
- Structures/ Signatures
- Control Structures

- As we talk about the above we'll continue to introduce ML.

More on Polymorphism

- We were talking about types, polymorphism and type inference.
- There are several kinds of polymorphism: for instance, function overloading is a kind of polymorphism (**ad hoc**); overriding methods in subclasses of classes is called **pure** or **subtype** polymorphism.
- ML's type inference scheme, where we define a function's type parameters implicitly, and then can use the function on any subtype of this type is called **implicit parametric polymorphism**.
- In contrast, things like C++ templates would be examples of **explicit parametric polymorphism**.
- In ML, example of **explicit parametric polymorphism** might be where we write a recursive data type like:

  datatype 'a Stack = EmptyStack | Stack of 'a*('a Stack);
- Here 'a is explicitly given as being of any type; we could then create value of types like int Stack. Ex: Stack(1, EmptyStack)

ML Modules

- ML provides a notion of unit data abstraction called a **module**. This is similar to a C++ namespace or Java Package.

```
structure SomeName = struct
   val bob =50;
   (* more values *)
   fun curry a b = (a, b);
   (* more functions *)
end;
```

- To use things in SomeName, I can either do things like:

```
SomeName.bob (* or *)
open SomeName;
bob;
```

Signatures

- Notice when we define/open a structure ML gives us its type back.
- ML also has a mechanism for creating types of modules called signatures:

```
signature SomeType = sig
    val bob:int;
    val curry : int -> int -> int * int
    (* last thing does not end with a ; *)
end;
```

- Notice, curry on the last slide was type 'a -> 'b -> 'a * 'b; but above is int -> int -> int * int.
- We can use SomeType to create narrowed instances of our previous structure, again illustrating parametric polymorphism:

```
structure SomeOtherName: SomeType = SomeName;
```

Some Useful ML Modules

- There are several useful modules which come with ML:
  - Int, Bool, Char, String each have the corresponding conversion functions for the given type
  - Math - has abs, sin, cos, tan, etc
  - Substring: has substring, splitl, splitr, triml, trimr, token, etc
  - TextIO - openIn, openOut, print, etc.
- Some function in the global environment are bound to things in these modules for example int actually binds to Int.int
- To find out more about these modules: http://www.standardml.org/Basis/overview.html

Control Structures

- Recall at the beginning of the semester we distinguished between two main kinds of abstractions connected with programming languages: **data abstraction** and **control abstraction**.
- We divided each of these into three levels: basic, structure and unit abstraction.
- We have now discussed in detail each of these levels for data abstractions, and gave examples of each in the ML language: primitive and enumerated type; type constructors and recursive type; and structures and signatures.
- We now begin our study of control abstractions looking at each of these levels in turn.

Basic Control Structures

- We first set up some terminology, which is often abused when people talk about particular languages:
  – A (pure) **expression** is a piece of code which executes some computation, returns a value, and has no side-effect (doesn't alter program memory).
  – A **statement** is a piece of code which is executed for its side-effect and which returns no value.
- We will now look at some of the control question which arise when we evaluate expressions and statements.

Expressions

- Depending on the language expressions can be written using infix (C, ML), prefix (Scheme), postfix (RPN calculators) notations. So (3 + 4)*5, might look like * + 3 4 5, or 3 4 + 5 *.
- + and * are called **operators**, the inputs they take are called **parameters/operands**, the particular values of those parameters in a given use of these operators are called **arguments**.
- There are several ways one could evaluate the arguments to expressions.
- We have seen **applicative evaluation**: compute the values of all subexpressions, then apply the root operator.
- For boolean expressions, one also has things like **short-circuit evaluation**: keep evaluating subexpression left to right until the value is determine then stop. Ex: 3 = 4 orelse 2= 0 orelse 1=1 orelse 1 = 0; (*would not bother to evaluate the last 1 = 0 *)
- We have also seen that **if-expressions** and **case-expressions** don't evaluate all their arguments. These are examples of **delayed evaluation.**

Normal Order Evaluation

- In this kind of evaluation, evaluation begins before its operands are evaluated, and each operand is evaluated only as needed.
- This is sometimes called **lazy-evaluation**.
- In a language with applicative order evaluation, one way to achieve normal order evaluation is to define for each argument to create a new argument which is its **delay**'d version.
- So for example, consider the function:

  fun count 0 b = b | count a b = count (a - 1) (b + 1);
- (count 1000000000 0) is slow enough you can notice it compute.
- Notice though: val b = fn () => count 100000000 0; computes instantaneously as the expression count 100000000 0 isn't being evaluated, b is just a function which if applied would compute count 100000000 0. We call b here is the delay of count 100000000 0;
- Once we have replaced each argument with its delayed, we rewrite our function to operate on delay'd argument.
- To evaluate a delayed argument (**force it**) we apply it on an empty argument list: For b, we do b();
- Now to do normal order evaluation, we take our rewritten function and make sure to only force argument as we need them.

Statements

- We have already seen several kinds of statements: if statements, case statements.
- There are also structured statements like: for, while, do-while loops.
- In ML, while loops can be done using the syntax like:

```
val i = ref 1;
while !i <= 10 do (print (Int.toString(!i)); print (" "); i := !i +1);
```

- You sometimes see the control of these statement written abstractly as: if B1 -> S1 | …| Bn ->Sn. Here Bi is called a **guard** on the statement Si.
- do-while can be viewed as just syntactic sugar on the basic while loop.
- Many languages restrict the way the control variables of a for loop work. For instance, i cannot be changed in the body of the loop, i is undefined after the loop terminates, i must be an int, etc.