CS 565: Programming Languages

Handout 1 January 11, 2009

Project

The programming assignments you will implement will be over a small subset of Standard ML called Mini-ML; your implementation will be in Standard ML. While Mini-ML shares strong syntactic and semantic similarity with SML, it differs in a number of important respects. When the semantics of an expression are underspecified in the documentation, it should be assumed that the semantics follows ML semantics for that expression.

Mini-ML is a higher-order mostly functional language that supports integer lists, recursive functions, pattern-matching using **case** expressions, tuples, records, and various unary and binary operations. Its syntax is identical to SML.

Programs

A Mini-ML program is a sequence of declarations and expressions. There are two kinds of declarations. A `val` declaration, `val x = e` evaluates expression `e` and binds the result to the variable `x`. A `fun` declaration,

`fun f(x:`*argType*`):`*resType* `= e`

declares a recursive function `f` whose result type is *resType* and which takes a single argument `x` having type *argType*. The body of the function is defined by expression `e`.

Each declaration makes its binding name visible to subsequent declarations in the program. For example, the following program defines variables `a` and `b`, a factorial function `fact`, and several function calls to `fact`:

```
val a = 10
val b = 4
fun fact(n:int):int =
  if n = 0
     then 1
     else n * fact(n-1)
val r1 = fact(a)
val r2 = fact(b)
val r3 = r1 + r2
```

Unlike SML, type declarations specifying the argument type and result type of a function are required. Also, Mini-ML does not support pattern-matching on function arguments. Thus, the following SML expression is ill-formed:

```
fun f(a:int, b:int):int = a + b
```

To get the effect of this function in Mini-ML, we would write the slightly more wordy:

```
fun f(args: (int * int)):int = (case args of
                                   (a,b) => a + b )
```

Primitive Types

Mini-ML supports integers, reals and strings as primitive types. For integers and reals, the following infix operations are provided:

```
+,-,*,~,>,>=,<,<=,=
```

Thus, for integers (reals) `a` and `b`,

- `~a` ⇒ negation of `a`
- `a` + `b` ⇒ sum of `a` and `b`
- `a - b` ⇒ difference of `a` and `b`
- `a` * `b` ⇒ product of `a` and `b`
- `a` *logical op* `b` ⇒ `true` if `a` *logical op* `b` holds for *logical op* ∈ {`<`,`>`,`<=`,`>=`,`=`} and `false` otherwise.

Mini-ML does not support implicit coercions between integers and reals. Thus, reals and integers cannot be supplied as different arguments to the same binary arithmetic or logical operation.

A string constant is written using quotations (e.g., "this is a string"). Strings can be concatenated using `^` thus: `"foo"^"bar"`.

Tuples and Records

Besides simple types, mini-ML also provides support for tuples and records. These types have a syntax and semantics similar to SML. Thus, the expression `(1,2.0,''foo'')` defines a three-tuple consisting of integer values `1,2.0` and `''foo''`. Elements of a tuple are extracted using pattern-matching (see below).

A record defines a collection of *label*, *value* pairs. The expression,

```
{x = 1, y = 2.0, z = "foo"}
```

defines a record consisting of fields `x`, `y` and `z` whose corresponding type is {`x:int,:real,z:string`}. Like SML, fields are extracted from a record thus:

```
let val r = { x = 1, y = 2.0, z = "foo" }
in #z r
end
```

Let-expressions

As in SML, local bindings can be declared using `let`-syntax. The expression: `let val x = b in e end` evaluates expression `b` to yield value *v*, and binds *v* to `x` and returns the result of evaluating expression `e` in this augmented context. Thus,

```
let val x = 2 + 3 in x + x end
```

yields `10`. Note that `x` is *not* visible in the evaluation of `b`.

Local recursive functions can be declared thus:

```
let fun f(x:int):int = if x = 0 then 1 else f(x-1) in f(3) end
```

Like SML, any pattern can appear on the left-hand side of a `let`-binding. Thus, the following expressions are also valid:

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

or

```
let val f = fn (x:int) => (x,3)
    val (x,y) = f(3)
in x + y  (* result is 6 *)
end
```

Unlike SML, Mini-ML does not support mutually-recursive function definitions.

Local functions and application

Like SML, functions are first-class objects in Mini-ML and can be declared within a local context. The expression:

```
fn (z:int) => z + 1
```

declares a function that adds one to its integer argument. Type annotations on arguments is required. No type annotations are necessary on the result type.

Because functions are first-class, they can take functional arguments. Thus, the following expression defines a function `twice`, which given a functional argument `f`, returns a function which given an integer argument `z` applies `f` twice to `z`. In the expression, `twice` is applied to a function that adds one to its argument. The function returned by `twice` in this example, when applied to 10, returns 12 as its result.

```
let val twice = fn (f:int -> int) => fn (z:int) => f(f(z))
in twice (fn (a:int) => a + 1) 10
end
```

Note that functional argument types are written using arrow notation: the type expression `t1 -> t2` defines a type describing functions that take `t1` arguments and returns `t2` results.

Case Expressions and Pattern Matching

Mini-ML supports a simple form of pattern-matching using `case` expressions. `Let`-expressions described above can be thought of a simple form of `case`; thus:

```
let p = e1 in e2 end
```

is equivalent to:

```
case e1 of
  p => e2
```

A subject defined as the first part of the `case` expression can *match* against a number of different target patterns defined as separate cases. The rules for matching are similar to SML. A subject pattern can be any expression. The value of this expression v is matched against the patterns defined by the targets. Patterns are evaluated top-down. Once a pattern successfully matches the subject, the corresponding match expression is evaluated, and returned as the value of the `case` expression. A successful match may bind variables in the target pattern to the values of the corresponding components in the subject.

For example, the following pattern defines a subject whose value is a tuple `(1,2)`, and a matching pattern that binds `x` to `2` in the corresponding match expression; the value of the entire `case` expression is `3`.

```
case (1,2) of
   (y,1) => y             (* does not match subject (1,2) *)
|  (1,x) => x + 1         (* matches subject *)
|  _ => 0                 (* will never be tested *)
```

The `_` defines a wildcard that matches any pattern. Unlike SML, pattern matching in mini-ML does not check for exhaustiveness or overlapping patterns.

Note the following use of `case` encodes simple `if-then-else` conditionals:

```
fn (x:bool) => case x of
                 true => true branch
               | false => false branch
```

Records can also be used in pattern matching expressions. Thus, the following expression yields the value of field `x` if field `z` is `true` and the value of field `y` otherwise.

```
fn (z:{x:int,y:int,z:bool}) =>
  case z of
    { x = x, y = y, z = c } => if c then x else y
```

Like SML, mini-ML does not support duplicate identifiers in record patterns. Unlike SML, string constants are not allowed on the right-hand side of a binding in a record pattern.

Datatypes and constructors

Mini-ML provides three built-in datatypes:

```
datatype bool = true | false
datatype intlist = Nil | Cons of (int * intlist)
datatype intoption = NONE | SOME of int
```

The `bool` datatype is obvious, and defines nullary constructors `true` and `false`. Integer lists are constructed using the `intlist` datatype. The following program defines a function `map` that that applies its function argument to all elements of its integer list argument. The function `iota` given an integer n returns a list containing all numbers from 0 to n.

```
fun map (f:int->int):intlist->intlist =
  (fn (l:intlist) =>
     case l of
      Nil => Nil
    | Cons (x,xs) => Cons (f(x), (map f) xs))

fun iotaHelper(x: (int * intlist)):intlist =
  (case x of
     (0,l) => l
   | (n,l) => iotaHelper(n-1,Cons(n,l)))

fun iota(n:int):intlist = iotaHelper(n,Nil)

val l = iota(3)

val m = map (fn (x:int) => x + 1) l
```

Assignment

Like SML, Mini-ML allows first-class references to be created, referenced, and assigned. The expression `ref x` returns a reference containing the value denoted by `x`. The expression `(!r)` returns the contents of location `r`; and the expression `r:=v` assigns the value `v` to the contents denoted by `r`.

Standard ML

We will use Standard ML as our implementation language. SML is a strongly-typed higher-order programming language with a rich module system. Each assignment will define a collection of structures (module implementations) and signatures (module specifications). We'll provide the relevant signatures for these modules; your job will be to provide implementations that conform to these signatures.

Documentation about the SML/NJ implementation can be found at
`http://www.smlnj.org/doc`. SML/NJ is installed on the machines in the Xinu lab. However, you are not obligated to use these machines; if you prefer to work on separate machines, download SML/NJ from
`http://www.smlnj.org/software.html`. Programming assignments will use the compilation manager facility (CM) of SML to organize signatures and implementations. For those of you unfamiliar with CM, details can be found at
`http://www.smlnj.org/doc/CM/index.html`

For those you who use Emacs, there is an *sml-mode* available that provides formatting, font highlighting, and an interface to the SML/NJ compiler. It is part of the emacs path on the Xinu machines, but for those of you who want to work on different machines, you can get sml-mode from
`http://www.smlnj.org/software.html`