Elements of Functional Programming

Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University
March, 2003

FUNCTIONAL PROGRAMMING

Characteristics of pure functional programming:
– Programming without assignments. The value of an expression depends only on the values of its subexpressions, if any.
– Implicit storage management. Storage is allocated as necessary by built-in operations on data. Storage that becomes inaccessible is automatically deallocated.
– Functions are first-class values. Functions have the same status as any other values. A function can be the value of an expression, it can be passed as an argument, and it can be put in a data structure.

COMPUTING WITH EXPRESSIONS

Example expressions:
2 An integer constant
x A variable
log n Function log applied to n
2+3 Function + applied to 2 and 3
Expressions can also include conditionals and function definitions.
if x ≥ y then x else y

A LITTLE LANGUAGE OF EXPRESSIONS

A quilt is:
– One of the primitive pieces (a) or (b)
– It is formed by turning a quilt clockwise 90º or
– It is formed by sewing a quilt to the right of another quilt of equal height
– Nothing else is a quilt

OPERATIONS ON QUILTS

QUILTS MADE UP OF SIMPLER PIECES

(a) (b)
Let a be \( a \) and b be \( b \). Then, the BNF syntax for quilt expressions is as follows:

\[
<\text{expression}> ::= a \mid b \mid \text{turn}\ (<\text{expression}> ) \mid \\
\text{sew}\ (<\text{expression}> <\text{expression}> )
\]

The semantics of an quilt expression specifies the quilt denoted by the expression \( \text{turn}\ (\text{sew}\ (\text{turn}\ (a) ), b) \) = \( \).

\[\text{untrun}\ (x) = \text{Turn 3 times}\ x\]

\[\text{fun}\ \text{unturn}\ (x) = \text{turn}\ (\text{turn}\ (\text{turn}\ (x)))\]

\[\text{fun}\ \text{pile}\ (x, y) = \text{turn}\ (\text{sew}\ (\text{untrun}\ (x), \text{untrun}\ (y)))\]

Once declared, a function can be used to declare others.

\[\text{let}\ \text{fun}\ \text{unturn}\ (x) = \text{turn}\ (\text{turn}\ (\text{turn}\ (x)))\ \text{in}\ \\
\text{fun}\ \text{pile}\ (x, y) = \text{turn}\ (\text{sew}\ (\text{untrun}\ (x), \text{untrun}\ (y)))\ \text{end}\]

Any other name can be used instead of \( x \) without changing the meaning of the expression.

\[\text{let}\ \text{val}\ \text{bnw} = \text{untrun}\ (b)\ \text{val}\ \text{bse} = \text{turn}\ (b)\ \text{in}\ \\
\text{pile}\ (\text{bnw}, \text{bse})\ \text{end}\]

The specification of the language Little Quilt can be found in Fig. 8.6. Its main constructs are borrowed from ML.
A type consists of a set of elements called values together with a set of functions called operations. Types are denoted by type expressions.

```
integer = { -2, -1, 0, 1, 2, ··· }
```

"2 is of type integer" means "2 ∈ { -2, -1, 0, 1, 2, ··· }"

Conventionally, [false, true] is boolean type.

Common categories of types:
- Basic Types
- Products of Types
- Lists of Elements
- Functions from a Domain to a Range

Basic values have no internal structure, so the only operation defined for all basic types is a comparison of equality.

For example, the equality 2=2 is true and the inequality 2≠2 is false.
**TYPE CONSTRUCTORS IN ML**

Type constructors (in order of increasing precedence):

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>function ( \rightarrow )</td>
<td>infix ( \text{int} \rightarrow \text{bool} )</td>
</tr>
<tr>
<td>product ( \times )</td>
<td>infix ( \text{int} \times \text{int} )</td>
</tr>
<tr>
<td>list ( \text{list} )</td>
<td>postfix ( \text{string list} )</td>
</tr>
</tbody>
</table>

**QUILTS IN ML**

A quilt is a list of rows.
A row is a list of squares.
A square has a texture and a direction.
Call the textures arcs and bands.
Call the directions ne, se, sw, and nw.

This view leads to the following representation:

```ml
datatype texture = arcs | bands;
datatype direction = ne | se | sw | nw;
type square = texture * direction;
type row = square list;
type quilt = row list;
```

**FUNCTION DECLARATIONS**

An expression is formed by applying a function or operation to subexpressions. Once a function is declared, it can be applied as an operator within expressions.

A function declaration has three parts:

- The name of the declared function
- The parameters of the function
- A rule for computing a result from the parameters

The basic syntax for function declaration is

```ml
fun <name> <formal-parameter> = <body>;
```

Example:

```ml
fun successor n = n + 1;
```

An alternative form:

```ml
fun successor(n) = n + 1;
```

The syntax for function application is

```ml<br>
<name> <actual-parameter>
```

Example:

```ml<br>successor(2+3)
```

**RECURSIVE FUNCTIONS**

A function \( f \) is recursive if its body contains an application of \( f \). More generally, a function \( f \) is recursive if \( f \) can activate itself, possibly through other functions.

Examples:

```ml
fun len(l) =
  if null(l) then 0 else 1 + len(tl(l));
```

```ml
fun fib(n) =
  if n=0 orelse n=1 then 1
  else fib(n-1) + fib(n-2);  
```

**INNERMOST EVALUATION**

Under the innermost-evaluation rule, the evaluation of a function application

```ml<br><name> <actual-parameter>
```

proceeds as follows:

- Evaluate the expression represented by \( \text{actual-parameter} \).
- Substitute the result for the formal in the function body.
- Evaluate the body.
- Return its value as the answer.

Each evaluation of a function body is called an activation of the function.

The approach of evaluating arguments before the function body is also referred to as call-by-value evaluation. Call-by-value can be implemented efficiently, so it is widely used. Under call-by-value, all arguments are evaluated, whether their values are needed or not.
SELECTIVE EVALUATION

The ability to evaluate selectively some parts of an expression and ignore others is provided by the construct

```
if <condition> then <expression1> else <expression2>;
```

OUTERMOST EVALUATION

Under the outermost-evaluation rule, the evaluation of a function application

```
<name> <actual-parameter>
```

proceeds as follows:

− Substitute the actual (without evaluating it) for the formal in the function body.
− Evaluate the body.
− Return its value as the answer.

Innermost and outermost evaluation produce the same result if both terminate with a result.

The distinguishing difference between the evaluation methods is that actual parameters are evaluated as they are needed in outermost evaluation; they are not evaluated before substitution.

Standard ML uses call-by-value or innermost evaluation.

SHORT-CIRCUIT EVALUATION

The operators andalso and orelse in ML perform short-circuit evaluation of boolean expressions, in which the right operand is evaluated only if it has to be.

E andalso F is false if E is false; it is true if both E and F are true. The evaluation of E andalso F proceeds from left to right, with F being evaluated only if E is true.

So E andalso F may terminate even if F does not.

The evaluation of E orelse F is true if E evaluates to true. F is skipped if E is true.

The 91-function

```
fun f(x) =
  if x > 100 then x-10 else f(f(x+11))
```

 Innermost Evaluation

```
f(100) = if 100>100 then 100-10 else f(f(100+11)) = f(f(111)) = f(if 111>100 then 111-10 else f(f(111+11))) = f(111-10) = f(101) = if 101>100 then 101-10 else f(f(101+11)) = 101-10 = 91
```

 Outermost Evaluation

```
f(100) = if 100>100 then 100-10 else f(f(100+11)) = f(f(100+11)) = if f(100+11)>100 then f(100+11)-10 else f(f(100+11)+11))
```

For simplicity, the next few lines show only the evaluation of (100+11):

```
f(100+11) = if 100+11>100 then 100+11-10 else f(f(100+11)+11)) = if 111>100 then 100+11-10 else f(f(100+11)+11))) = 100+11-10 = 111-10 = 101
```

Returning to the evaluation of f(100):

```
f(100) = if 100>100 then (100+11)-10 else f(f(100+11)+11)) = f(100+11)-10 = 111-10 = 101
```

OUTERMOST vs. INNERMOST

“Outermost” appears to do more work than “innermost”. “Outermost” can terminate where “innermost” fails.

```
fun or(x,y) =
  if x then true else y;
```

− Under innermost-evaluation, both subexpressions E and F in or (E,F) are evaluated before they are substituted into the function body. So or (true,F) results in a nonterminating computation if the evaluation of F does not terminate.

− Under outermost-evaluation,

```
  or (true,F) = if true then true else F;
```

so the computation terminates regardless of the evaluation of F terminates or not.

Since ML uses innermost evaluation, the operator orelse has to be provided by the language. It cannot be user-defined as part of a program.
LEXICAL SCOPE

Bound occurrences of variables can be renamed without changing the meaning of a program. For example,

```
fun successor(n) = n + 1;
```

This renaming principle is the basis for the lexical scope rule for determining the meanings of names in programs.

```
fun addy(x) = x + y;
```

What is y? When a function declaration refers to a name that is not a formal parameter, the value of that name has to be determined by some context.

Lexical scope rules use the program text surrounding a function declaration to determine the context in which nonlocal names are evaluated. The program text is static in contrast to run-time execution, so such rules are also called static scope rules.

VAL BINDINGS

The occurrence of x to the right of keyword val in

```
let val x = E_1 in E_2 end
```

is called a binding occurrence or simply binding of x. All occurrences of x in E_1 are said to be within the scope of this binding; the scope of a binding includes itself.

The occurrences of x within the scope of a binding are said to be bound. A binding of a name is said to be visible to all occurrences of the name in the scope of the binding.

VAL BINDINGS (cont.)

```
let val x = 2 in let val x = x+1 in x*x end end
```

The value of an expression is left undisturbed if we replace all occurrences of a variable x within the scope of a binding of x by a fresh variable.

```
let val x = 2 in
let val y = x+1 in y*y end end
```

FUN BINDINGS

The occurrences of f and x to the right of keyword fun in

```
let fun f(x) = E_1 in E_2 end
```

are bindings of f and x. This binding of the formal parameter x is visible only to the occurrences of x in E_1.

This binding of the function name f is visible to the occurrences of f in both E_1 and E_2.

```
let fun f(x) = x+1 in f(x) end end
```

NESTED BINDINGS

Sequences of val and fun bindings are treated as nested bindings. Thus,

```
let val x_1 = E_1
         x_2 = E_2

in E end
```

is treated as if the individual bindings were nested:

```
let val x_1=E_1 in let val x_2=E_2 in E end end
```

This approach generalizes to any sequence of val and fun bindings.

SIMULTANEOUS BINDINGS

Mutually recursive functions require the simultaneous binding of more than one function name. In

```
let fun f(x) = E_1 and f'(x) = E_2

in E end
```

the scope of both f_1 and f_2 includes E_1, E_2, and E. The scopes of the formal parameters x_1 and x_2 are, as usual, limited to the respective function bodies.

```
let fun even(x)=
    if x=0 then true else
    if x div 2 = 0 then false else odd(x-1)

and odd(x)=
    if x=1 then false else even(x-1)

in (even(24), odd(24)) end
```
**TYPE CHECKING**

Type distinctions between values carry over to expressions. A type system for a language is a set of rules for associating a type with expressions in the language. A type system rejects an expression if it does not associate a type with the expression.

**Why we need a type system for each language?**
To detect program errors as early as possible.

Wherever possible, ML infers the type of an expression. An error is reported if the type of the expression cannot be inferred.

At the heart of all type systems is the following rule for function application:

\[ \text{If } f \text{ is a function of type } A \rightarrow B, \text{ and } a \text{ has type } A, \text{ then } f(a) \text{ has type } B. \]

**TYPE EQUIVALENCE**

Two type expressions are structurally equivalent if and only if they are equivalent under the following rules:

- A type name is structurally equivalent to itself.
- Two type expressions are structurally equivalent if they are formed by applying the same type constructor to structurally equivalent types.
- After a type declaration, \( n = T \), the type name \( n \) is structurally equivalent to \( T \).

ML uses structural equivalence of types.

\[
\begin{align*}
&[\text{arcs,ne}] \\
&\text{val it} = [\text{arcs,ne}] : \text{(texture * direction) list list}
\end{align*}
\]

The type of this expression is structurally equivalent to the type name \( \text{quilt} \) declared as follows:

\[
\begin{align*}
&\text{type square} = \text{texture*direction} \\
&\text{type row} = \text{square list} \\
&\text{type quilt} = \text{row list}
\end{align*}
\]

**OVERLOADING**

A symbol is overloaded if it has different meanings in different contexts. Familiar operator symbols like + and * are overloaded.

When ML cannot resolve overloading, it complains, as in

\[
\text{fun add}(x,y) = x + y; \\
\text{stdIn:10.17 Error: overloaded variable cannot be resolved: +}
\]

Explicit types can then be given to resolve overloading.

\[
\begin{align*}
&\text{fun add}(x,y):\text{int} = x + y; \\
&\text{val add} = \text{fn} : \text{int * int -> int}
\end{align*}
\]

- \( \text{fun add}(x,y) = x + \text{int}; \)
- \( \text{val add} = \text{fn} : \text{int * int -> int} \)

**COERCION: IMPLICIT TYPE CONVERSION**

A coercion is a conversion from one type to another, inserted automatically by a programming language.

\[
\begin{align*}
&2 * 3.142; \\
&\text{stdIn:1.1 Error: expected integer type, found type: real}
\end{align*}
\]

Type conversions must be specified explicitly in ML because the language does not coerce types.

\[
\begin{align*}
&\text{real}(2); \\
&\text{val it} = 2.0 : \text{real}
\end{align*}
\]

**POLYMORPHISM: PARAMETERIZED TYPES**

For all lists, the function \( \text{hd} \) returns the head or first element of a list:

\[
\begin{align*}
&\text{hd} [1,2,3]; \\
&\text{val it} = 1 : \text{int}
\end{align*}
\]

- \( \text{hd} ["a", "b", "c"] \);
- \( \text{val it} = \text{"a"} : \text{string} \)

What is the type of \( \text{hd} \)?

\[
\begin{align*}
&\text{hd}; \\
&\text{val it} = \text{fn : a list -> a}
\end{align*}
\]

ML uses a leading quote, as in \( \text{a} \), to identify a type parameter. ML is known for its support for polymorphic functions, which can be applied to parameters of more than one type.