Program Design with ML Types and Pattern Matching
(Plus Syntax Help)

Norman Ramsey

Spring 2019

1 Introduction

This handout sketches how to transfer your design method to ML, a language with types and pattern matching. Popular languages with these features also include Haskell, Elm, OCaml, Reason, F♯, Erlang, and Scala. More obscure languages include Agda, Idris, and Coq/Gallina. The handout also provides a little syntax help.

2 Design steps

Our design method is affected by the introductions of *constructed data* and *types*.

1. *Forms of data* for numbers and functions are as in Scheme. But forms of data for lists, pairs, tuples, trees, and other constructed data are determined by primitive types and user-defined types, including algebraic datatypes. These forms are shown in Table 1, as *patterns*.

   Patterns are the technical name for the phrases that appear as function arguments on the left-hand sides of algebraic laws—so you already know how to program with them. But to define them carefully, here are ML's rules for patterns:

   - Any variable, as in `x`, is a pattern.
   - The "wildcard," as in `_` (underscore), is a pattern
   - A sequence of patterns separated by commas and wrapped in round brackets, as in `(l, x, r)`, is a *tuple* pattern.
   - The *empty tuple* `()` is a pattern.
   - A sequence of patterns separated by commas and wrapped in square brackets, as in `[x1, x2, x3]`, is a *list* pattern.
   - The *empty list* `[]` is a pattern.
   - A value constructor by itself, as in `nil` or `NONE`, is a pattern.
   - A *value constructor* applied to a pattern, as in `SOME x`, is a pattern.
   - An *infix* value constructor placed between patterns, as in `x :: xs`, is a pattern.[1]
   - A sequence of pattern bindings separated by commas and wrapped in curly brackets, as in `{ ps1 = s, ps2 = s' }`, is a *record* pattern. These are rare.
   - A literal number, as in `2018`, is a pattern.
   - A literal string, as in `"frogs"`, is a pattern.
   - A literal character, as in `#"f"`, is a pattern.

   A key feature of ML is that you get to define new forms of data, using the `datatype` definition. For example, a binary tree:

   ```
   datatype 'a tree
      = LEAF
      | NODE of 'a tree * 'a * 'a tree
   ```

2. *Example inputs* include what you would expect from μScheme: numbers written using numeric literals, and anonymous lambda functions written using `fn`, as in `(fn (x, y) => y + 1)`. ML also has string literals.

   In addition, examples of constructed data are formed using the pattern rules: if a pattern has no variables or wildcards, it specifies a value. Examples:

   ```
   (105, "hello")
   [2, 3, 5, 7, 11]
   SOME 33
   ```

3. *Function names* are limited. In ML, you may use *either* "symbolic" characters like `<`, `?`, `+`, and so on, *or* you may use alphanumeric characters[2] with an underscore, but you may not use both in the same name. Symbolic characters may be combined into arbitrarily long names, such as `<=>` or `/*/`.

[1] Confusingly, "fixity" is a local property of a name, not a property of the value constructor itself.

[2] The ASCII quote mark, here pronounced "prime," counts as alphanumeric, as in `x'`, pronounced "x-prime."

Type of e	*Patterns*	*Test in definition*	*Test in expression*	*Types of parts*
`'a list`	`[]` `x :: xs`	`fun f []` = ⋯ `  | f (x :: xs)` = ⋯	`case` *e* `  of []` => ⋯ `  | x :: xs` => ⋯	`x : 'a, xs : 'a list`
`'a option`	`NONE` `SOME x`	`fun f NONE` = ⋯ `  | f (SOME x)` = ⋯	`case` *e* `  of NONE` => ⋯ `  | SOME x` => ⋯	`x : 'a`

`order`	`LESS`
	`EQUAL`
	`GREATER`
		`fun f LESS` = ⋯
		`  | f EQUAL` = ⋯
		`  | f GREATER` = ⋯
			`case` *e*
			`  of LESS` => ⋯
			`  of EQUAL` => ⋯
			`  of GREATER` => ⋯	

`int`	`0` `n`	`fun f 0` = ⋯ `  | f n` = ⋯	`case` *e* `  of 0` => ⋯ `  | n` => ⋯	`n : int`
`'a * 'b`	`(x, y)`	`fun f (x, y)` = ⋯	`let val (x, y)` = *e* `in` ⋯ `end`	`x : 'a, y : 'b`
`'a * 'b * 'c`	`(x, y, z)`	`fun f (x, y, z)` = ⋯	`let val (x, y, z)` = *e* `in` ⋯ `end`	`x : 'a, y : 'b, z : 'c`

`{ f1 : 'a`
`, f2 : 'b`
`, f3 : 'c`
` ⋯`
`}`
(record)
	`{ f1 = x`
	`, f2 = y`
	`, f3 = z`
	`, ...`
	`}`
	(“`f1`” is short for “field 1”, and so on)
		`fun f { f1 = x`
		`  , f2 = y`
		`  , f3 = z`
		`  , ...`
		`  } = ⋯`
			`let val { f1 = x`
			`  , f2 = y`
			`  , f3 = z`
			`  , ... } = e`
			`in` ⋯
			`end`
				`x : 'a`
				`y : 'b`
				`z : 'c`

`'a tree` (homework)	`LEAF` `NODE(l,x,r)`	`fun f (LEAF)` = ⋯ `  | f (NODE(l,x,r))` = ⋯	`case` *e* `  of LEAF` => ⋯ `  | NODE(l,x,r)` => ⋯	`l : 'a tree, x : 'a` `r : 'a tree`

μScheme
`exp`
(page 366)
	`LITERAL v`
	`VAR x`
	`SET (x, e)`
	⋮
		`fun f (LITERAL v)` = ⋯
		`  | f (VAR x)` = ⋯
		`  | f (SET (x, e))` = ⋯
		⋮
			`case` *e*
			`  of LITERAL v` => ⋯
			`  | VAR x` => ⋯
			`  | SET (x, e)` => ⋯
			⋮
				`v : value`
				`x : name`
				`x : name, e : exp`
				⋮

Table 1: Identifying forms and extracting parts (ML builtins and 105 types)

ML offers a design choice not available in Scheme: function *names* can be "infix." Predefined functions like `mod`, `o`, and `+` all have infix names, as does the value constructor `::`. The fixity of names can be changed by an `infix` or `nonfix` definition form. It's especially common for symbolic names to be made infix.

Infix names like `::` and `+` can't be used as first-class values; when you write an infix name, ML thinks you mean to apply it. But there is a workaround: putting the reserved word `op` in front of any infix name turns it into a nonfix name, which you can use as a value. Here are two classic examples:

```
fun sum  ns = foldl op + 0 ns
fun prod ns = foldl op * 1 ns
```

4. *Function contracts* are now enhanced: every function has a *most general type*. We consider the type to be part of the function's contract. The type is enforced by the compiler. In many cases, like `List.all` and `List.exists`, the name and the type form a sufficient contract all by themselves.

5. *Example results* and test cases work using the same ideas as in μScheme ("check-expect," "check-assert," and "check-error"), but the mechanisms are different. On the minus side, ML doesn't have linguistic support for unit testing, so we wind up writing unit tests using anonymous `(fn =>` $\cdots$`)`, and we have to supply our own string-conversion functions. On the plus side, ML indicates checked run-time errors using *exceptions*, and it's possible to check for the presence of a particular exception, like `Subscript`, `Empty`, or `Overflow`.

6. *Algebraic laws* are as helpful as ever. They must respect types. We will also develop a new design method that helps with writing right-hand sides of algebraic laws. The new method is based on types, and when we are ready to study types in detail, it will be presented in class.

7. *Coding case analysis* is *much* simpler than in Scheme: for case analysis over constructed data, we just use pattern matching. This feature makes the code look an awful lot like the algebraic laws. For case analysis of natural numbers or machine integers, we can often use partial pattern matching (one or more cases of interest, followed by a catchall case).

8. *Coding right-hand sides* uses the same design methods. But in ML, both the concrete syntax and the abstract syntax are different from Scheme. Here are the key differences in the abstract syntax and our use of it:

   - In ML, the `let` form uses definitions and has a similar semantics to Scheme's `let*`. Direct recursion is accomplished by using `fun`, and mutual recursion by using `and`.
   - ML has a `case` form for pattern matching in an expression. (But usually we will pattern match in the `fun` definition form.)
   - Deconstruction of input data is *always* done by pattern matching. ML has functions like `car`, `cdr`, and `null?`, but they are used rarely, and only by experts.

9. *Revisiting tests* has the same intellectual content, but it's much more fussy to code. To run your tests, you'll need to study the `Unit` interface that is described in the guide to learning ML.

3 Syntax help

Standard ML is a full language, not simplified for teaching, and an exhaustive syntax summary would be overwhelming. This section presents the key syntactic tools that you will use most frequently. It is not exhaustive!

ML has four major syntactic categories. From the top down:

d	Definitions
p	Patterns
e	Expressions
τ	Types

These categories are related like this:

- *Definitions* sit at the top, and they contain both patterns and expressions. A typical definition form has a pattern on the left and an expression on the right. The definition of a Curried function may have multiple patterns on the left.

  The `val` form you already know is present, but instead of just a name on the left, it can take any pattern.

  The `define` form you already know is a special case of `fun`, but `fun` is more typically used with patterns, to express algebraic laws directly.

  There are two kinds of *type definitions*: type abbreviations (`type`) and fresh, algebraic data types (`datatype`). Both are called "types," and both definition forms contain types.

- *Patterns* are new. They are one of the two main interesting features of ML, and they are described in detail above. Patterns may contain types, but they usually don't—we put types in patterns only when we're debugging.

- *Expressions* resemble those that you already know, except for the `let` form. ML's `let` form contains *definitions*, and it has the same semantics as Scheme's `let*`.

  Expressions may contain definitions and types. Expressions commonly contain definitions (any `let` form), but they rarely contain types—we put types in expressions only when we're debugging.

- *Types* are more general than the types you know from C and C++. We will study types at length.

To summarize the common forms of the categories listed above, I use these symbols for nonterminals:

x, f	Name (of a variable or function)
k	Literal (like 7 or `#"a"`)
K	Name of a value constructor
t	Name of a type

Using these symbols, here are some examples of the most commonly used forms of ML syntax:

$p \Rightarrow x \mid k \mid$ `(`p_1`,` p_2`)` $\mid$ `(`p_1`,` p_2`,` p_3`)`
$\mid$ `[]` $\mid p_1$ `::` p_2 $\mid$ `[`p_1`,`p_2`, ...,` p_n`]`
$\mid$ `NONE` $\mid$ `SOME` p $\mid$ `LESS` $\mid$ `EQUAL` $\mid$ `GREATER`
$\mid K \mid K\ p$

$e \Rightarrow x \mid k \mid$ `(`e_1`,` e_2`)` $\mid$ `(`e_1`,` e_2`,` e_3`)`
$\mid$ `[]` $\mid e_1$ `::` e_2 $\mid$ `[`e_1`,`e_2`, ...,` e_n`]`
$\mid$ `NONE` $\mid$ `SOME` e $\mid$ `LESS` $\mid$ `EQUAL` $\mid$ `GREATER`
$\mid K \mid K\ e$

$\mid\ e\ e \cdots$
$\mid$ `if` e_1 `then` e_2 `else` e_3
$\mid$ `let` $d \cdots$ `in` e `end`
$\mid$ `(case` e `of` p_1 `=>` e_1 `|` p_2 `=>` e_2 `|` $\cdots$`)`
$\mid$ `raise` $e \mid$ `(`e_1 `handle` p `=>` e`)`
$\mid$ e_1 `andalso` $e_2 \mid e_1$ `orelse` e_2

$d \Rightarrow$ `val` p `=` e
$\mid$ `val (`x_1`,` x_2`)` `=` e (overlooked special case)
$\mid$ `fun` $f\ p_1$ `=` e_1 `|` $f\ p_2$ `=` e_2 `|` $\cdots$
$\mid$ `fun` $f\ p_1 \cdots$ `=` e_1 `|` $f\ p_2 \cdots$ `=` e_2 `|` $\cdots$
$\mid$ `exception` K
$\mid$ `exception` K `of` τ
$\mid$ `type` t `=` τ
$\mid$ `type 'a` t `=` τ
$\mid$ `datatype` t `=` K_1 `of` τ_1 `|` K_2 `of` τ_2 `|` $\cdots$
$\mid$ `datatype` t `=` K_1 `|` $\cdots$
$\mid$ `datatype 'a` t `=` K_1 `of` τ_1 `|` K_2 `of` τ_2 `|` $\cdots$
$\mid$ `datatype 'a` t `=` K_1 `|` $\cdots$

$\tau \Rightarrow$ `int` $\mid$ `string` $\mid$ `bool` $\mid$ `char`
$\mid\ \tau$ `list` $\mid\ \tau$ `option`
$\mid\ \tau_1$ `*` $\tau_2 \mid \tau_1$ `*` τ_2 `*` τ_3
$\mid\ \tau_1$ `->` τ_2
$\mid$ `'a` $\mid$ `'b` $\mid$ `'c`