4 Case Expressions and Pattern Matching

Earlier we gave several examples of pattern matching in defining functions---for example `length` and `fringe`. In this section we will look at the pattern-matching process in greater detail (§3.17). (Pattern matching in Haskell is different from that found in logic programming languages such as Prolog; in particular, it can be viewed as "one-way" matching, whereas Prolog allows "two-way" matching (via unification), along with implicit backtracking in its evaluation mechanism.)

Patterns are not "first-class;" there is only a fixed set of different kinds of patterns. We have already seen several examples of *data constructor* patterns; both `length` and `fringe` defined earlier use such patterns, the former on the constructors of a "built-in" type (lists), the latter on a user-defined type (`Tree`). Indeed, matching is permitted using the constructors of any type, user-defined or not. This includes tuples, strings, numbers, characters, etc. For example, here's a contrived function that matches against a tuple of "constants:"

```
contrived :: ([a], Char, (Int, Float), String, Bool) -> Bool
contrived    ([],  'b',  (1,   2.0),   "hi",   True) = False
```

This example also demonstrates that *nesting* of patterns is permitted (to arbitrary depth).

Technically speaking, *formal parameters* (The Report calls these *variables*.) are also patterns---it's just that they *never fail to match a value*. As a "side effect" of the successful match, the formal parameter is bound to the value it is being matched against. For this reason patterns in any one equation are not allowed to have more than one occurrence of the same formal parameter (a property called *linearity* §3.17, §3.3, §4.4.3).

Patterns such as formal parameters that never fail to match are said to be *irrefutable*, in contrast to *refutable* patterns which may fail to match. The pattern used in the `contrived` example above is refutable. There are three other kinds of irrefutable patterns, two of which we will introduce now (the other we will delay until Section 4.4).

As-patterns.

Sometimes it is convenient to name a pattern for use on the right-hand side of an equation. For example, a function that duplicates the first element in a list might be written as:

```
f (x:xs)               = x:x:xs
```

(Recall that ":" associates to the right.) Note that `x:xs` appears both as a pattern on the left-hand side, and an expression on the right-hand side. To improve readability, we might prefer to write `x:xs` just once, which we can achieve using an *as-pattern* as follows: (Another advantage to doing this is that a naive implementation might completely reconstruct `x:xs` rather than re-use the value being matched against.)

```
f s@(x:xs)              = x:s
```

Technically speaking, as-patterns always result in a successful match, although the sub-pattern (in this case `x:xs`) could, of course, fail.

Wild-cards.

Another common situation is matching against a value we really care nothing about. For example, the functions `head` and `tail` defined in Section 2.1 can be rewritten as:

```
head (x:_)              = x
tail (_:xs)             = xs
```

in which we have "advertised" the fact that we don't care what a certain part of the input is. Each wild-card independently matches anything, but in contrast to a formal parameter, each binds nothing; for this reason more than one is allowed in an equation.

4.1 Pattern-Matching Semantics

So far we have discussed how individual patterns are matched, how some are refutable, some are irrefutable, etc. But what drives the overall process? In what order are the matches attempted? What if none succeeds? This section addresses these questions.

Pattern matching can either *fail*, *succeed* or *diverge*. A successful match binds the formal parameters in the pattern. Divergence occurs when a value needed by the pattern contains an error (_|_). The matching process itself occurs "top-down, left-to-right." Failure of a pattern anywhere in one equation results in failure of the whole equation, and the next equation is then tried. If all equations fail, the value of the function application is _|_, and results in a run-time error.

For example, if `[1,2]` is matched against `[0,bot]`, then `1` fails to match `0`, so the result is a failed match. (Recall that `bot`, defined earlier, is a variable bound to _|_.) But if `[1,2]` is matched against `[bot,0]`, then matching `1` against `bot` causes divergence (i.e. _|_).

The other twist to this set of rules is that top-level patterns may also have a boolean *guard*, as in this definition of a function that forms an abstract version

of a number's sign:

```
sign x |  x >  0        =   1
       |  x == 0        =   0
       |  x <  0        =  -1
```

Note that a sequence of guards may be provided for the same pattern; as with patterns, they are evaluated top-down, and the first that evaluates to `True` results in a successful match.

4.2 An Example

The pattern-matching rules can have subtle effects on the meaning of functions. For example, consider this definition of `take`:

```
take  0     _          =  []
take  _     []         =  []
take  n     (x:xs)     =  x : take (n-1) xs
```

and this slightly different version (the first 2 equations have been reversed):

```
take1  _     []        =  []
take1  0     _         =  []
take1  n    (x:xs)     =  x : take1 (n-1) xs
```

Now note the following:

```
take  0 bot  => []
take1 0 bot  => _|_
take  bot []  => _|_
take1 bot []  => []
```

We see that `take` is "more defined" with respect to its second argument, whereas `take1` is more defined with respect to its first. It is difficult to say in this case which definition is better. Just remember that in certain applications, it may make a difference. (The Standard Prelude includes a definition corresponding to `take`.)

4.3 Case Expressions

Pattern matching provides a way to "dispatch control" based on structural properties of a value. In many circumstances we don't wish to define a *function* every time we need to do this, but so far we have only shown how to do pattern matching in function definitions. Haskell's *case expression* provides a way to solve this problem. Indeed, the meaning of pattern matching in function definitions is specified in the Report in terms of case expressions, which are considered more primitive. In particular, a function definition of the form:

`f` p_{11} ... p_{1k} `=` e_1

...
f p_{n1} ... p_{nk} = e_n

where each p_{ij} is a pattern, is semantically equivalent to:

```
f x1 x2 ... xk = case (x1, ..., xk) of
(p11, ..., p1k) -> e1
...
(pn1, ..., pnk) -> en
```

where the `xi` are new identifiers. (For a more general translation that includes guards, see §4.4.3.) For example, the definition of `take` given earlier is equivalent to:

```
take m ys               = case (m,ys) of
                            (0,_)       ->  []
                            (_,[])      ->  []
                            (n,x:xs)    ->  x : take (n-1) xs
```

A point not made earlier is that, for type correctness, the types of the right-hand sides of a case expression or set of equations comprising a function definition must all be the same; more precisely, they must all share a common principal type.

The pattern-matching rules for case expressions are the same as we have given for function definitions, so there is really nothing new to learn here, other than to note the convenience that case expressions offer. Indeed, there's one use of a case expression that is so common that it has special syntax: the *conditional expression*. In Haskell, conditional expressions have the familiar form:

```
if e1 then e2 else e3
```

which is really short-hand for:

```
case e1 of True  -> e2
           False -> e3
```

From this expansion it should be clear that e_1 must have type `Bool`, and e_2 and e_3 must have the same (but otherwise arbitrary) type. In other words, `if-then-else` when viewed as a function has type `Bool->a->a->a`.

4.4 Lazy Patterns

There is one other kind of pattern allowed in Haskell. It is called a *lazy pattern*, and has the form ~pat. Lazy patterns are *irrefutable*: matching a value v against

~pat always succeeds, regardless of pat. Operationally speaking, if an identifier in pat is later "used" on the right-hand-side, it will be bound to that portion of the value that would result if v were to successfully match pat, and _|_ otherwise.

Lazy patterns are useful in contexts where infinite data structures are being defined recursively. For example, infinite lists are an excellent vehicle for writing *simulation* programs, and in this context the infinite lists are often called *streams*. Consider the simple case of simulating the interactions between a server process `server` and a client process `client`, where `client` sends a sequence of *requests* to `server`, and `server` replies to each request with some kind of *response*. This situation is shown pictorially in Figure 2. (Note that `client` also takes an initial message as argument.)

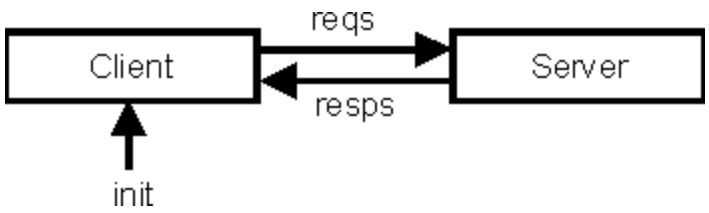

Figure 2

Using streams to simulate the message sequences, the Haskell code corresponding to this diagram is:

```
reqs                    = client init resps
resps                   = server reqs
```

These recursive equations are a direct lexical transliteration of the diagram.

Let us further assume that the structure of the server and client look something like this:

```
client init (resp:resps) = init : client (next resp) resps
server      (req:reqs)   = process req : server reqs
```

where we assume that `next` is a function that, given a response from the server, determines the next request, and `process` is a function that processes a request from the client, returning an appropriate response.

Unfortunately, this program has a serious problem: it will not produce any output! The problem is that `client`, as used in the recursive setting of `reqs` and `resps`, attempts a match on the response list before it has submitted its first request! In other words, the pattern matching is being done "too early." One way to fix this is to redefine `client` as follows:

```
client init resps         = init : client (next (head resps)) (tail resps)
```

Although workable, this solution does not read as well as that given earlier. A

better solution is to use a lazy pattern:

```
client init ~(resp:resps) = init : client (next resp) resps
```

Because lazy patterns are irrefutable, the match will immediately succeed, allowing the initial request to be "submitted", in turn allowing the first response to be generated; the engine is now "primed", and the recursion takes care of the rest.

As an example of this program in action, if we define:

```
init                    = 0
next resp               = resp
process req             = req+1
```

then we see that:

```
take 10 reqs => [0,1,2,3,4,5,6,7,8,9]
```

As another example of the use of lazy patterns, consider the definition of Fibonacci given earlier:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

We might try rewriting this using an as-pattern:

```
fib@(1:tfib)    = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

This version of `fib` has the (small) advantage of not using `tail` on the right-hand side, since it is available in "destructured" form on the left-hand side as `tfib`.

[This kind of equation is called a *pattern binding* because it is a top-level equation in which the entire left-hand side is a pattern; i.e. both `fib` and `tfib` become bound within the scope of the declaration.]

Now, using the same reasoning as earlier, we should be led to believe that this program will not generate any output. Curiously, however, it *does*, and the reason is simple: in Haskell, pattern bindings are assumed to have an implicit ~ in front of them, reflecting the most common behavior expected of pattern bindings, and avoiding some anomalous situations which are beyond the scope of this tutorial. Thus we see that lazy patterns play an important role in Haskell, if only implicitly.

4.5 Lexical Scoping and Nested Forms

It is often desirable to create a nested scope within an expression, for the purpose of creating local bindings not seen elsewhere---i.e. some kind of "block-structuring" form. In Haskell there are two ways to achieve this:

Let Expressions.

Haskell's *let expressions* are useful whenever a nested set of bindings is required. As a simple example, consider:

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

The set of bindings created by a `let` expression is *mutually recursive*, and pattern bindings are treated as lazy patterns (i.e. they carry an implicit ~). The only kind of declarations permitted are *type signatures*, *function bindings*, and *pattern bindings*.

Where Clauses.

Sometimes it is convenient to scope bindings over several guarded equations, which requires a *where clause*:

```
f x y  |  y>z           =  ...
       |  y==z          =  ...
       |  y<z           =  ...
     where z = x*x
```

Note that this cannot be done with a `let` expression, which only scopes over the expression which it encloses. A `where` clause is only allowed at the top level of a set of equations or case expression. The same properties and constraints on bindings in `let` expressions apply to those in `where` clauses.

These two forms of nested scope seem very similar, but remember that a `let` expression is an *expression*, whereas a `where` clause is not---it is part of the syntax of function declarations and case expressions.

4.6 Layout

The reader may have been wondering how it is that Haskell programs avoid the use of semicolons, or some other kind of terminator, to mark the end of equations, declarations, etc. For example, consider this `let` expression from the last section:

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

How does the parser know not to parse this as:

```
let y   = a*b f
    x   = (x+y)/y
in f c + f d
```

?

The answer is that Haskell uses a two-dimensional syntax called *layout* that

essentially relies on declarations being "lined up in columns." In the above example, note that `y` and `f` begin in the same column. The rules for layout are spelled out in detail in the Report (§2.7, §B.3), but in practice, use of layout is rather intuitive. Just remember two things:

First, the next character following any of the keywords `where`, `let`, or `of` is what determines the starting column for the declarations in the where, let, or case expression being written (the rule also applies to `where` used in the class and instance declarations to be introduced in Section 5). Thus we can begin the declarations on the same line as the keyword, the next line, etc. (The `do` keyword, to be discussed later, also uses layout).

Second, just be sure that the starting column is further to the right than the starting column associated with the immediately surrounding clause (otherwise it would be ambiguous). The "termination" of a declaration happens when something appears at or to the left of the starting column associated with that binding form. (Haskell observes the convention that tabs count as 8 blanks; thus care must be taken when using an editor which may observe some other convention.)

Layout is actually shorthand for an *explicit* grouping mechanism, which deserves mention because it can be useful under certain circumstances. The `let` example above is equivalent to:

```
let { y   = a*b
    ; f x = (x+y)/y
    }
in f c + f d
```

Note the explicit curly braces and semicolons. One way in which this explicit notation is useful is when more than one declaration is desired on a line; for example, this is a valid expression:

```
let y   = a*b;  z = a/b
    f x = (x+y)/z
in f c + f d
```

For another example of the expansion of layout into explicit delimiters, see §2.7.

The use of layout greatly reduces the syntactic clutter associated with declaration lists, thus enhancing readability. It is easy to learn, and its use is encouraged.
