# Chapter 6

# Case Analysis

## 6.1 Homogeneous and Heterogeneous Types

Tuple types have the property that all values of that type have the same form ($n$-tuples, for some $n$ determined by the type); they are said to be *homogeneous*. For example, all values of type `int*real` are pairs whose first component is an integer and whose second component is a real. Any type-correct pattern will match any value of that type; there is no possibility of failure of pattern matching. The pattern `(x:int,y:real)` is of type `int*real` and hence will match any value of that type. On the other hand the pattern `(x:int,y:real,z:string)` is of type `int*real*string` and cannot be used to match against values of type `int*real`; attempting to do so *fails at compile time*.

Other types have values of more than one form; they are said to be *heterogeneous* types. For example, a value of type `int` might be `0`, `1`, `~1`, ... or a value of type `char` might be `#"a"` or `#"z"`. (Other examples of heterogeneous types will arise later on.) Corresponding to each of the values of these types is a pattern that matches only that value. Attempting to match any other value against that pattern *fails at execution time* with an error condition called a *bind failure*.

Here are some examples of pattern-matching against values of a heterogeneous type:

```
val 0 = 1-1
val (0,x) = (1-1, 34)
val (0, #"0") = (2-1, #"0")
```

The first two bindings succeed, the third fails. In the case of the second, the variable `x` is bound to `34` after the match. No variables are bound in the first or third examples.

## 6.2 Clausal Function Expressions

The importance of constant patterns becomes clearer once we consider how to define functions over heterogeneous types. This is achieved in ML using a *clausal function expression* whose general form is

fn $pat_1$ => $exp_1$
  | $\vdots$
  | $pat_n$ => $exp_n$

Each $pat_i$ is a pattern and each $exp_i$ is an expression involving the variables of the pattern $pat_i$. Each component *pat*=>*exp* is called a *clause*, or a *rule*. The entire assembly of rules is called a *match*.

The typing rules for matches ensure consistency of the clauses. Specifically, there must exist types $typ_1$ and $typ_2$ such that

1. Each pattern $pat_i$ has type $typ_1$.
2. Each expression $exp_i$ has type $typ_2$, given the types of the variables in pattern $pat_i$.

If these requirements are satisfied, the function has the type $typ_1$->$typ_2$.

Application of a clausal function to a value *val* proceeds by considering the clauses *in the order written*. At stage $i$, where $1 \leq i \leq n$, the argument value *val* is matched against the pattern $pat_i$; if the pattern match succeeds, evaluation continues with the evaluation of expression $exp_i$, with the variables of $pat_i$ replaced by their values as determined by pattern matching. Otherwise we proceed to stage $i + 1$. If no pattern matches (*i.e.*, we reach stage $n + 1$), then the application fails with an execution error called a *match failure*.

Here's an example. Consider the following clausal function:

```
val recip : int -> int =
    fn 0 => 0 | n:int => 1 div n
```

This defines an integer-valued reciprocal function on the integers, where the reciprocal of `0` is arbitrarily defined to be `0`. The function has two clauses, one for the argument `0`, the other for non-zero arguments `n`. (Note that `n` is guaranteed to be non-zero because the patterns are considered in order: we reach the pattern `n:int` only if the argument fails to match the pattern `0`.)

The `fun` notation is also generalized so that we may define `recip` using the following more concise syntax:

```
fun recip 0 = 0
  | recip (n:int) = 1 div n
```

One annoying thing to watch out for is that the `fun` form uses an equal sign to separate the pattern from the expression in a clause, whereas the `fn` form uses a double arrow.

Case analysis on the values of a heterogeneous type is performed by application of a clausally-defined function. The notation

```
case exp
  of pat1 => exp1
   | ...
   | patn => expn
```

is short for the application

```
(fn pat1 => exp1
  | ...
  | patn => expn)
exp.
```

Evaluation proceeds by first evaluating *exp*, then matching its value successively against the patterns in the match until one succeeds, and continuing with evaluation of the corresponding expression. The `case` expression fails if no pattern succeeds to match the value.

## 6.3 Booleans and Conditionals, Revisited

The type `bool` of booleans is perhaps the most basic example of a heterogeneous type. Its values are `true` and `false`. Functions may be defined on booleans using clausal definitions that match against the patterns `true` and `false`.

For example, the negation function may be defined clausally as follows:

```
fun not true = false
  | not false = true
```

The conditional expression

```
if exp then exp1 else exp2
```

is short-hand for the case analysis

```
case exp
  of true => exp1
   | false => exp2
```

which is itself short-hand for the application

```
(fn true => exp1 | false => exp2) exp.
```

The "short-circuit" conjunction and disjunction operations are defined as follows. The expression *exp*$_1$ `andalso` *exp*$_2$ is short for

```
if exp1 then exp2 else false
```

and the expression *exp*$_1$ `orelse` *exp*$_2$ is short for

```
if exp1 then true else exp2.
```

You should expand these into case expressions and check that they behave as expected. Pay particular attention to the evaluation order, and observe that the call-by-value principle is not violated by these expressions.

## 6.4 Exhaustiveness and Redundancy

Matches are subject to two forms of "sanity check" as an aid to the ML programmer. The first, called *exhaustiveness checking*, ensures that a well-formed match *covers* its domain type in the sense that every value of the domain must match one of its clauses. The second, called *redundancy checking*, ensures that no clause of a match is subsumed by the clauses that precede it. This means that the set of values covered by a clause in a match must not be contained entirely within the set of values covered by the preceding clauses of that match.

Redundant clauses are *always* a mistake — such a clause can never be executed. Redundant rules often arise accidentally. For example, the second rule of the following clausal function definition is redundant:

```
fun not True = false
  | not False = true
```

By capitalizing `True` we have turned it into a variable, rather than a constant pattern. Consequently, *every* value matches the first clause, rendering the second redundant.

Since the clauses of a match are considered in the order they are written, redundancy checking is correspondingly order-sensitive. In particular, changing the order of clauses in a well-formed, irredundant match can make it redundant, as in the following example:

```
fun recip (n:int) = 1 div n
  | recip 0 = 0
```

The second clause is redundant because the first matches *any* integer value, including `0`.

Inexhaustive matches may or may not be in error, depending on whether the match might ever be applied to a value that is not covered by any clause. Here is an example of a function with an inexhaustive match that is plausibly in error:

```
fun is_numeric #"0" = true
    | is_numeric #"1" = true
    | is_numeric #"2" = true
    | is_numeric #"3" = true
    | is_numeric #"4" = true
    | is_numeric #"5" = true
    | is_numeric #"6" = true
    | is_numeric #"7" = true
    | is_numeric #"8" = true
    | is_numeric #"9" = true
```

When applied to, say, `#"a"`, this function fails. Indeed, the function never returns `false` for any argument!

Perhaps what was intended here is to include a *catch-all* clause at the end:

```
fun is_numeric #"0" = true
    | is_numeric #"1" = true
    | is_numeric #"2" = true
    | is_numeric #"3" = true
    | is_numeric #"4" = true
    | is_numeric #"5" = true
    | is_numeric #"6" = true
    | is_numeric #"7" = true
    | is_numeric #"8" = true
    | is_numeric #"9" = true
    | is_numeric _ = false
```

The addition of a final catch-all clause renders the match exhaustive, because any value not matched by the first ten clauses will surely be matched by the eleventh.

Having said that, it is a very bad idea to simply add a catch-all clause to the end of every match to suppress inexhaustiveness warnings from the compiler. The exhaustiveness checker is your friend! Each such warning is a suggestion to double-check that match to be sure that you've not made a silly error of omission, but rather have intentionally left out cases that are ruled out by the invariants of the program. In chapter 10 we will see that the exhaustiveness checker is an extremely valuable tool for managing code evolution.

## 6.5 Sample Code

Here is the complete code for this chapter.

# Chapter 7

# Recursive Functions

So far we've only considered very simple functions (such as the reciprocal function) whose value is computed by a simple composition of primitive functions. In this chapter we introduce *recursive* functions, the principal means of iterative computation in ML. Informally, a recursive function is one that computes the result of a call by possibly making further calls to itself. Obviously, to avoid infinite regress, some calls must return their results without making any recursive calls. Those that do must ensure that the arguments are, in some sense, "smaller" so that the process will eventually terminate.

This informal description obscures a central point, namely the means by which we may convince ourselves that a function computes the result that we intend. In general we must prove that for all inputs of the domain type, the body of the function computes the "correct" value of result type. Usually the argument imposes some additional assumptions on the inputs, called the *pre-conditions*. The correctness requirement for the result is called a *post-condition*. Our burden is to prove that for every input satisfying the pre-conditions, the body evaluates to a result satisfying the post-condition. In fact we may carry out such an analysis for many different pre- and post-condition pairs, according to our interest. For example, the ML type checker proves that the body of a function yields a value of the range type (if it terminates) whenever it is given an argument of the domain type. Here the domain type is the pre-condition, and the range type is the post-condition. In most cases we are interested in deeper properties, examples of which we shall consider below.

To prove the correctness of a recursive function (with respect to given pre- and post-conditions) it is typically necessary to use some form of inductive reasoning. The base cases of the induction correspond to those cases that make no recursive calls; the inductive step corresponds to those that do. The beauty of inductive reasoning is that we may *assume* that the recursive calls work correctly when showing that a case involving recursive calls is correct. We must separately show that the base cases satisfy the given pre- and post-conditions. Taken together, these two steps are sufficient to establish the correctness of the function itself, by appeal to an induction principle that justifies the particular pattern of recursion.

No doubt this all sounds fairly theoretical. The point of this chapter is to show that it is also profoundly practical.

## 7.1 Self-Reference and Recursion

In order for a function to "call itself", it must have a name by which it can refer to itself. This is achieved by using a *recursive value binding*, which are ordinary value bindings qualified by the keyword `rec`. The simplest form of a recursive value binding is as follows:

`val rec` *var*`:`*typ* `=` *val*.

As in the non-recursive case, the left-hand is a pattern, but here the right-hand side must be a value. In fact the right-hand side must be a function expression, since only functions may be defined recursively in ML. The function may refer to itself by using the variable *var*.

Here's an example of a recursive value binding:

```
val rec factorial : int->int =
    fn 0 => 1 | n:int => n * factorial (n-1)
```

Using `fun` notation we may write the definition of factorial much more clearly and concisely as follows:

```
fun factorial 0 = 1
  | factorial (n:int) = n * factorial (n-1)
```

There is obviously a close correspondence between this formulation of `factorial` and the usual textbook definition of the factorial function in terms of recursion equations:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad (n > 0) \end{aligned}$$

Recursive value bindings are type-checked in a manner that may, at first glance, seem paradoxical. To check that the binding

```
val rec var : typ = val
```

is well-formed, we ensure that the value *val* has type *typ*, *assuming* that *var* has type *typ*. Since *var* refers to the value *val* itself, we are in effect assuming what we intend to prove while proving it!

(Incidentally, since *val* is required to be a function expression, the type *typ* will always be a function type.)

Let's look at an example. To check that the binding for factorial given above is well-formed, we *assume* that the variable factorial has type int->int, then *check* that its definition, the function

```
fn 0 => 1 | n:int => n * factorial (n-1),
```

has type int->int. To do so we must check that each clause has type int->int by checking for each clause that its pattern has type int and that its expression has type int. This is clearly true for the first clause of the definition. For the second, we assume that n has type int, then check that n * factorial (n-1) has type int. This is so because of the rules for the primitive arithmetic operations and because of our assumption that factorial has type int->int.

How are applications of recursive functions evaluated? The rules are almost the same as before, with one modification. We must arrange that all occurrences of the variable standing for the function are replaced by the function itself before we evaluate the body. That way all references to the variable standing for the function itself are indeed references to the function itself!

Suppose that we have the following recursive function binding

```
val rec var : typ =
    fn pat_1 => exp_1
     | ...
     | pat_n => exp_n
```

and we wish to apply *var* to the value *val* of type *typ*. As before, we consider each clause in turn, until we find the first pattern $pat_i$ matching *val*. We proceed, as before, by evaluating $exp_i$, replacing the variables in $pat_i$ by the bindings determined by pattern matching, but, *in addition*, we replace all occurrences of the *var* by its binding in $exp_i$ before continuing evaluation.

For example, to evaluate `factorial 3`, we proceed by retrieving the binding of `factorial` and evaluating

```
(fn 0=>1 | n:int => n*factorial(n-1))(3).
```

Considering each clause in turn, we find that the first doesn't match, but the second does. We therefore continue by evaluating its right-hand side, the expression `n * factorial(n-1)`, after replacing `n` by `3` and `factorial` by its definition. We are left with the sub-problem of evaluating the expression

```
3 * (fn 0 => 1 | n:int => n*factorial(n-1))(2)
```

Proceeding as before, we reduce this to the sub-problem of evaluating

```
3 * (2 * (fn 0=>1 | n:int => n*factorial(n-1))(1)),
```

which reduces to the sub-problem of evaluating

```
3 * (2 * (1 * (fn 0=>1 | n:int => n*factorial(n-1))(0))),
```

which reduces to

```
3 * (2 * (1 * 1)),
```

which then evaluates to `6`, as desired.

Observe that the repeated substitution of `factorial` by its definition ensures that the recursive calls really do refer to the factorial function itself. Also observe that the size of the sub-problems grows until there are no more recursive calls, at which point the computation can complete. In broad outline, the computation proceeds as follows:

1. `factorial 3`
2. `3 * factorial 2`

3. `3 * 2 * factorial 1`

4. `3 * 2 * 1 * factorial 0`

5. `3 * 2 * 1 * 1`

6. `3 * 2 * 1`

7. `3 * 2`

8. `6`

Notice that the size of the expression first grows (in direct proportion to the argument), then shrinks as the pending multiplications are completed. This growth in expression size corresponds directly to a growth in run-time storage required to record the state of the pending computation.

## 7.2 Iteration

The definition of `factorial` given above should be contrasted with the following two-part definition:

```
fun helper (0,r:int) = r
  | helper (n:int,r:int) = helper (n-1,n*r)
fun factorial (n:int) = helper (n, 1)
```

First we define a "helper" function that takes two parameters, an integer argument and an *accumulator* that records the running partial result of the computation. The idea is that the accumulator re-associates the pending multiplications in the evaluation trace given above so that they can be performed prior to the recursive call, rather than after it completes. This reduces the space required to keep track of those pending steps. Second, we define `factorial` by calling `helper` with argument `n` and initial accumulator value `1`, corresponding to the product of zero terms (empty prefix).

As a matter of programming style, it is usual to conceal the definitions of helper functions using a `local` declaration. In practice we would make the following definition of the iterative version of `factorial`:

```
local
      fun helper (0,r:int) = r
        | helper (n:int,r:int) = helper (n-1,n*r)
in
      fun factorial (n:int) = helper (n,1)
end
```

This way the helper function is not visible, only the function of interest is "exported" by the declaration.

The important thing to observe about `helper` is that it is *iterative*, or *tail recursive*, meaning that the recursive call is the last step of evaluation of an application of it to an argument. This means that the evaluation trace of a call to `helper` with arguments `(3,1)` has the following general form:

1. `helper (3, 1)`
2. `helper (2, 3)`
3. `helper (1, 6)`
4. `helper (0, 6)`
5. `6`

Notice that there is no growth in the size of the expression because there are no pending computations to be resumed upon completion of the recursive call. Consequently, there is no growth in the space required for an application, in contrast to the first definition given above. Tail recursive definitions are analogous to loops in imperative languages: they merely iterate a computation, without requiring auxiliary storage.

## 7.3 Inductive Reasoning

Time and space usage are important, but what is more important is that the function compute the intended result. The key to the correctness of a recursive function is an inductive argument establishing its correctness. The critical ingredients are these:

1. An *input-output specification* of the intended behavior stating *pre-conditions* on the arguments and a *post-condition* on the result.

2. A proof that the specification holds for each clause of the function, *assuming* that it holds for any recursive calls.

3. An *induction principle* that justifies the correctness of the function as a whole, given the correctness of its clauses.

We'll illustrate the use of inductive reasoning by a graduated series of examples. First consider the simple, non-tail recursive definition of `factorial` given in section 7.1. One reasonable specification for `factorial` is as follows:

1. *Pre-condition*: $n \geq 0$.

2. *Post-condition*: `factorial` $n$ evaluates to $n!$.

We are to establish the following statement of correctness of `factorial`:

> if $n \geq 0$, then `factorial` $n$ evaluates to $n!$.

That is, we show that the pre-conditions imply the post-condition holds of the result of any application. This is called a *total correctness* assertion because it states not only that the post-condition holds of any result of application, but, moreover, that every application in fact yields a result (subject to the pre-condition on the argument).

In contrast, a *partial correctness* assertion does not insist on termination, only that the post-condition holds whenever the application terminates. This may be stated as the assertion

> if $n \geq 0$ and `factorial` $n$ evaluates to $p$, then $p = n!$.

Notice that this statement is true of a function that diverges whenever it is applied! In this sense a partial correctness assertion is weaker than a total correctness assertion.

Let us establish the total correctness of `factorial` using the pre- and post-conditions stated above. To do so, we apply the principle of *mathematical induction* on the argument $n$. Recall that this means we are to establish the specification for the case $n = 0$, and, assuming it to hold for $n >= 0$, show that it holds for $n + 1$. The base case, $n = 0$, is trivial: by definition `factorial` $n$ evaluates to 1, which is $0!$. Now suppose that $n = m + 1$ for some $m >= 0$. By the inductive hypothesis we have that `factorial` $m$ evaluates to $m!$ (since $m \geq 0$), and so by definition `factorial` $n$ evaluates to

$$\begin{aligned} n \times m! &= (m+1) \times m! \\ &= (m+1)! \\ &= n!, \end{aligned}$$

as required. This completes the proof.

That was easy. What about the iterative definition of `factorial`? We focus on the behavior of `helper`. A suitable specification is given as follows:

1. *Pre-condition*: $n \geq 0$.

2. *Post-condition*: `helper (`$n$`, `$r$`)` evaluates to $n! \times r$.

To show the total correctness of `helper` with respect to this specification, we once again proceed by mathematical induction on $n$. We leave it as an exercise to give the details of the proof.

With this in hand it is easy to prove the correctness of `factorial` — if $n \geq 0$ then `factorial` $n$ evaluates to the result of `helper (n, 1)`, which evaluates to $n! \times 1 = n!$. This completes the proof.

Helper functions correspond to lemmas, main functions correspond to theorems. Just as we use lemmas to help us prove theorems, we use helper functions to help us define main functions. The foregoing argument shows that this is more than an analogy, but lies at the heart of good programming style.

Here's an example of a function defined by *complete* induction (or *strong* induction), the Fibonacci function, defined on integers $n >= 0$:

```
(* for n>=0, fib n yields the nth Fibonacci number *)
fun fib 0 = 1
  | fib 1 = 1
  | fib (n:int) = fib (n-1) + fib (n-2)
```

The recursive calls are made not only on `n-1`, but also `n-2`, which is why we must appeal to complete induction to justify the definition. This definition of `fib` is very inefficient because it performs many redundant computations: to compute `fib n` requires that we compute `fib (n-1)` and `fib (n-2)`. To compute `fib (n-1)` requires that we compute `fib (n-2)` a second time, and `fib (n-3)`. Computing `fib (n-2)` requires computing `fib` `(n-3)` again, and `fib (n-4)`. As you can see, there is considerable redundancy here. It can be shown that the running time `fib` of is exponential in its argument, which is quite awful.

Here's a better solution: for each $n >= 0$ compute not only the $n$th Fibonacci number, but also the $(n-1)$st as well. (For $n = 0$ we define the "$-1$st" Fibonacci number to be zero). That way we can avoid redundant recomputation, resulting in a linear-time algorithm. Here's the code:

```
(* for n>=0, fib' n evaluates to (a, b), where
   a is the nth Fibonacci number, and
   b is the (n-1)st *)
fun fib' 0 = (1, 0)
  | fib' 1 = (1, 1)
  | fib' (n:int) =
    let
        val (a:int, b:int) = fib' (n-1)
    in
        (a+b, a)
    end
```

You might feel satisfied with this solution since it runs in time linear in $n$. It turns out (see Graham, Knuth, and Patashnik, *Concrete Mathematics* (Addison-Wesley 1989) for a derivation) that the recurrence

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

has a closed-form solution over the real numbers. This means that the $n$th Fibonacci number can be calculated directly, without recursion, by using floating point arithmetic. However, this is an unusual case. In most instances recursively-defined functions have no known closed-form solution, so that some form of iteration is inevitable.

## 7.4 Mutual Recursion

It is often useful to define two functions simultaneously, each of which calls the other (and possibly itself) to compute its result. Such functions are said to be *mutually recursive*. Here's a simple example to illustrate the point, namely testing whether a natural number is odd or even. The most obvious approach is to test whether the number is congruent to 0 mod 2, and indeed this is what one would do in practice. But to illustrate the idea of mutual recursion we instead use the following inductive characterization: 0 is even, and not odd; $n > 0$ is even iff $n - 1$ is odd; $n > 0$ is odd iff $n - 1$ is even. This may be coded up using two mutually-recursive procedures as follows:

```
fun even 0 = true
  | even n = odd (n-1)
and odd 0 = false
  | odd n = even (n-1)
```

Notice that `even` calls `odd` and `odd` calls `even`, so they are not definable separately from one another. We join their definitions using the keyword `and` to indicate that they are defined simultaneously by mutual recursion.

## 7.5 Sample Code

Here is the complete code for this chapter.