Thus entering a function's name without any parameters is the equivalent of entering a value. However, the major difference between a function value and other values is that two functions may not be tested for equality. This is the case even if they both have precisely the same code or precisely the same mappings for all possible input values. Thus the expression `(dd = double)` will result in an error.

Defining functions

In Miranda, new functions are introduced in three steps:

1. Declare the function name and its type (its argument and result types):
   ```
   square ::  num -> num
   ```
2. Provide the appropriate pre- and post-conditions:
   ```
   ||pre:  none
   ||post: square n  = n^2
   ```
3. Describe the function using one or more equations:
   ```
   square n = n*n
   ```

Although type declarations are not mandatory for functions, it is good programming practice to include them with definitions in all programs. Type declarations act as a design aid since the programmer is forced to consider the nature of the input and output of functions before defining them. They also document the program and make it more readable since any programmer can immediately see what types of objects are mapped by the function. Of course, the second step is also optional in that the evaluator won't even notice if you miss it out. But we hope by now you are beginning to understand why it is essential.

Consider quadratic equations of the form $ax^2 + bx + c = 0$, where x is a variable and a, b and c are constants. Now the solutions for such a quadratic equation are given by

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We can define a function `hasSolutions` which given `a`, `b` and `c` returns `True` or `False` indicating whether there will be any solution for `x`:

```
hasSolutions:: num -> num -> num -> bool
||pre: none
||post: hasSolutions a b c iff a*x*x + b*x +c = 0 for some real x
hasSolutions a b c = ((a~=0) & (b*b>=4*a*c)) \/
                     ((a=0) & ((b~=0)\/(c=0)))
```

This uses the fact that the roots of the quadratic equation are given by the formula above.

NOTE:

- The specification is quite different from the definition, and it takes some mathematical reasoning to relate the two.
- `(a~=0 & b*b>=4*a*c) \/ ((a=0) & ((b~=0) \/ (c=0)))`, the right-hand side of the definition, has type `bool` and its value is exactly the Boolean result you want for the function application.

In the above definition `a b c` are called the *formal parameters*. We talk about the left-hand or the right-hand side of an *equation* or *rule*. The right-hand side describes how the result is constructed using the parameters.

Layout — the offside rule

Miranda assumes that the *entire* right-hand side of an equation lies *directly below or to the right* of the *first* symbol of the right-hand side. This enables the evaluator to spot automatically when the right-hand side of a rule has finished. An advantage of this is that no special character or symbol such as a semi-colon is required to indicate the end of definition — less typing for the programmer! This is possible because as soon as the evaluator comes across a symbol that violates the offside rule it will take the violation to mean that the right-hand side of the definition has been completed. On the negative side, however, care must be taken by the programmer to use safe layout. For long definitions leave a blank line before starting the right-hand side and indent a small standard amount. For example,

```
functionWithALongName =
    xxxx
```

or

```
functionWithALongName
    = xxxx
```

Remember that the boundary is set by the first symbol of the right-hand side and *not* by the preceding `=`.

4.4 More constructions

Case analysis

Often, we want to define a function by case analysis. For example,

```
pdifference ::  num -> num -> num
||pre:   none
||post:  pdifference x y = abs (x-y)
pdifference x y = x-y,   if x>=y
                = y-x,   if y>x
```

This definition is a *single* equation consisting of two expressions, each of which is distinguished by a Boolean-valued expressions called a *guard*. The first alternative says that the value of `(pdifference x y)` is `x-y` provided that the expression `x>=y` evaluates to `True`; `pdifference` is defined for all numbers since the two guards *exhaust* all possibilities. In the above the order in which the alternatives are written is *not* significant because the two cases are *disjoint* (that is, the guards are *mutually exclusive*), they can't both succeed. However, if cases are *not* disjoint then the order in which the alternatives are written *is* significant.

Thus guards allow us to choose between two or more alternative values of the same type and only one alternative will be selected and evaluated. If there is a possibility of more than one guard evaluating to `True`, then the alternative selected will be the *first* whose guard evaluates to `True`. Actually, it is good programming practice to write order-independent code, so it is better if guards are mutually exclusive. Also, writing order-independent code aids in the portability of your program: then your program is more like a set of equations. For example, if your guards are mutually exclusive then porting your Miranda program to a parallel machine in which guards may be evaluated simultaneously will not require any alterations to your code.

An equivalent definition for `pdifference` is

```
pdifference x y = x-y,   if x >= y
                = y-x,   otherwise
```

The reserved word `otherwise` can be regarded as a convenient abbreviation for the condition which returns `True` when all previous guards return `False`.

Pattern matching on basic types

Pattern matching is one of the more powerful features of functional languages. As we shall see in Chapter 6, it is most powerful when used with composite structures such as lists because it lets you delve into the structure. With the basic types it can still be used, though it tends to appear much like case analysis. The idea is that the formal parameters are not just variables, but 'patterns' to be matched against the actual parameter. For example,

```
bitNegate :: num -> num
||pre:  x = 0 \/ x = 1
||post: (x = 0 & b = 1) \/ (x = 1 & b = 0)
||       where b= bitNegate x
bitNegate 0 = 1
bitNegate 1 = 0
```

Thus pattern matching can be used to select amongst alternative defining equations of a function based on the format of the actual parameter. This facility has a number of advantages, including enhancing program readability and providing an alternative to the use of guards, which are inflexible at times. Furthermore, pattern matching often helps the programmer when considering all possible inputs to a function. For example, it is clear from the above equations that `bitNegate` is currently only defined for the values `0` and `1`.

The notions of disjointedness and exhaustiveness apply to patterns just as for guards; similarly, for non-disjoint patterns, it is the *first* match that is used. The `otherwise` guard corresponds to a final pattern that is simply a variable (and so matches everything). Note that pattern matching and guards can be used together:

```
sign ::  num -> num
||pre:  none
||post: (n=0 & sign n=0) \/
||      (n>0 & sign n=1) \/ (n<0 & sign n = -1)
sign 0  = 0
sign n  = 1,   if n>0
        = -1,  if n<0
```

Special facilities for pattern matching on natural numbers

Patterns can be used to define functions which operate on natural numbers (that is, non-negative integers). The operator `+` is special as it can be used in patterns of the form `p+k` where `p` is a pattern and `k` is a positive integer constant. A number `x` will match the pattern only if `x` is an integer and `x` $\geq$ `k`. For example, `y+1` matches any positive integer, and `y` gets bound to that integer-minus-one. So,

```
pred :: num -> num
||pre:  nat(x)
||post: (pred x = 0 & x = 0) \/ (x > 0 & pred x = x-1)
pred 0 = 0
pred (n + 1) = n
```

(`nat(x)` means that n is a natural number: $int(x) \wedge x \geq 0$.) Notice that patterns can contain variables. This definition describes a version of the predecessor function. The pattern `n+1` can only be 'matched' by a value if

`n` matches a natural number forcing `pred` to be defined for natural numbers only. Here the patterns are exhaustive and hence cover all natural numbers. Furthermore, we know that the order of equations will not be important in this example since the patterns are disjoint as no natural number can match more than one pattern.

Prefix and infix functions

In Miranda, enclosing an infix operator in parentheses converts it to an ordinary prefix function, which can be applied to its arguments like any other function. This can be useful in the context of Chapter 8, where functions are used as arguments of other functions:

```
Miranda  (+) 8 9
17

Miranda  (>) 8 9
False
```

Conversely, user-defined binary functions can also be applied in an infix form by prefixing their name with the special character `$`:

```
Miranda  9 $smaller 8
8
```

One simple way of determining whether it is a good idea to have an operator as an infix one is to see if it is associative — `(x $f y) $f z = x $f(y $f z)` This is because `x $f y $f z` is then unambiguous.

Local definitions

In mathematical descriptions one often finds expressions that are qualified by a phrase of the form 'where ...'. The same device can also be used in function definitions. For example, `balance*i where i = interestRate/100`. In fact, we have already used `where` in the definition of `fourthroot` in Chapter 3. The special reserved word `where` can be used to introduce local definitions whose context (or scope) is the expression on the entire right-hand side of the definition which contains it. For example,

```
f x y = x + a,   if x > 10
      = x - a,   otherwise
           where a = square (y+1)
```

In any one equation the `where` clause is written after the last alternative.

Its local definitions govern the whole of the right-hand side of that equation, including the guards, but do not apply to any other equation.

Furthermore, following a `where` there can be any number of definitions. These definitions are just like ordinary definitions and may therefore contain nested `where`s or be recursive definitions.

Note that the whole of the `where` clause must be indented, to show that it is part of the right-hand side of the equation. The evaluator determines the scopes of nested `where`s by looking at their indentation levels. In the next example it is clear that the definition of `g` is not local to the right-hand side of the definition of `f`, but those of `y` and `z` are

```
f x   = g y z
          where y = (x+1) * 4
                z = (x-1) * x
g x z = (x + 1) * (z-1)
```

Let us consider some uses of local definitions. Firstly, as in `fourthroot`, they can be used to avoid repeated evaluation. In an expression a subexpression may appear several times, for example

```
z+(smaller x y)*(smaller x y)
```

Here the subexpression `(smaller x y)` appears twice, and will be evaluated twice, which is rather wasteful. By using a local definition we can give a name to an expression and then use the name in the same way that we use a formal parameter:

```
z+w*w where w = smaller x y
```

If you like, you can view this use of local definitions as a mechanism for extending the existing set of formal parameters.

Local definitions can also be used to decompose compound structures or user-defined data types by providing names for components (as will be seen later, in Chapter 7).

It is good programming practice to avoid unnecessary nesting of definitions. In particular, use local definitions only if logically necessary. Furthermore, a third level of definition should be used only very occasionally. Failure to follow these simple programming guidelines will result in definitions that are difficult to read, understand and reason about.

4.5 Summary

- Miranda has three primitive data types: numbers, truth-values and characters (`num`, `bool` and `char` respectively).

- Miranda also provides many built-in operators and functions.
- A new function is defined in three stages. The function's type is declared, the function is specified in a comment and then it is defined using one or more equations.
- Although type declarations and specifications are not mandatory for functions, it is good programming practice to include them with all definitions.
- Miranda is layout-sensitive in that it assumes that the *entire* right-hand side of an equation lies *directly below or to the right* of the *first* symbol of the right-hand side (excluding the initial `=`). This is the *offside rule*.
- To aid in the portability of programs try, wherever possible, to write order-independent code. This means writing mutually exclusive guards or patterns.
- Functions (or other values) can also be defined *locally* to a definition. Such local definitions can be used to avoid repeated evaluation or to decompose compound structures, as will be seen in Chapter 7.

4.6 Exercises

1. Write definitions for the functions specified in the exercises at the end of Chapter 3.
2. Define `istriple`, which returns whether the sum of the squares of two numbers is equal to the square of a third number. A *Pythagorean triple* is a triple of whole numbers x, y and z that satisfy $x^2 + y^2 = z^2$. The Miranda function `istriple` should be declared as follows:

   ```
   istriple :: num -> num -> num -> bool
   ||pre:  none
   ||post: (istriple a b c) <-> a,b,c are the lengths of the
   ||                           sides of a right angle triangle
   ```

   The function takes as arguments three numbers and returns true if they form such a triple. Evaluate the function on the triples

   ```
   3 4 5
   5 12 13
   12 14 15
   ```

   and check that the first two are Pythagorean triples and the third is not. Do this exercise twice: first assume that c is the hypotenuse and then rewrite it so that any of the parameters could be the hypotenuse.

Chapter 5

Recursion and induction

5.1 Recursion

Suppose we want to write a function `sum` n which gives us the sum of the natural numbers up to n, that is, $\sum_{i=0}^{n} i$:

`sum` $n = 0 + 1 + 2 + 3 + \ldots + (n-1) + n$

Inspecting the above expression we see that if we remove '$+n$' we obtain an expression which is equivalent to `sum`$(n-1)$, at least if $n \geq 1$.

This suggests that

$$\texttt{sum}\ n = \texttt{sum}\ (n-1) + n \tag{5.1}$$

We say that the equation exhibits a *recurrence relationship*. To complete the definition we must define a *base case* which specifies where the *recursion* process should end. For `sum` this is when the argument is 0. Thus the required definition is

```
sum ::  num -> num
||pre:  nat(n)
||post: sum n = sum(i=0 to n) i
sum n   = 0,              if n = 0
        = sum (n-1) + n,  if n > 0
```

'`sum(i=0 to n) i`' is intended to be a typewriter version of '$\sum_{i=0}^{n} i$'. If we just used the recurrence relation (5.1), forgetting the base case, then we would obtain non-terminating computations as illustrated in Figure 5.1. Function definitions, like that of `sum`, that call themselves are said to be *recursive*. Obviously, the computation of sum involves repetition of an action.

Often when describing a function — such as `sum` — there are infinitely many cases to consider. In conventional imperative programming languages this is solved by using a *loop*, but in functional languages there are no explicit looping constructs. Instead, solutions to such problems are expressed

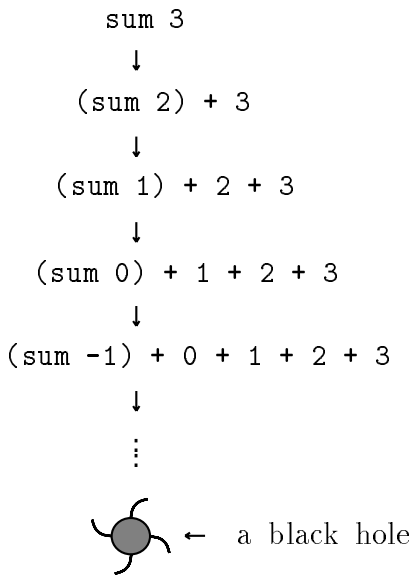

Figure 5.1

by defining a *recursive* function. Clearly, the recursive call must be in terms of a simpler problem — otherwise the recursion will proceed forever.

The example given above illustrated the technique of writing recursive functions, which can be summarised as follows:

1. Define the base case(s).
2. Define the recursive case(s):
   (a) reduce the problem to simpler cases of the same problem,
   (b) write the code to solve the simpler cases,
   (c) combine the results to give required answer.

5.2 Evaluation strategy of Miranda

We have seen that evaluation is a simple process of *substitution* and *simplification*, using primitive and user-defined function definitions. More precisely, a function application is rewritten (reduced) in two steps. First the actual parameters are substituted for the formal parameters in the defining equation of the function: this is called *instantiation*. Then the application is replaced by the instantiated right-hand side expression (see Figure 5.2).

During evaluation an expression may contain more than one *redex* — place where reduction is possible. But in functional languages if an expression has a well-defined value then the final result is independent of the reduction route (this is known as the *Church-Rosser property*). However, an evaluator selects

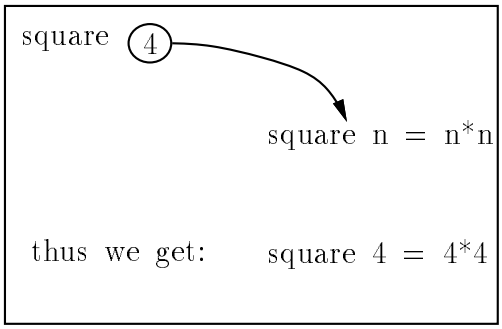

Figure 5.2

the next reduction (from the set of possible ones) in a consistent way. This is called the evaluator's *reduction strategy*. We will not discuss reduction strategies here except to mention that Miranda's reduction strategy is called *lazy evaluation*. Lazy evaluation works as follows:

Reduce a particular part *only* if its result is needed.

Therefore, because of lazy evaluation you can write function definitions such as

```
f n = 1,      if n = 0
    = n * y,  otherwise
      where y = f(n-1)
```

Although the scope of the local definition of `y` is the entire right-hand side of the equation for `f`, we know that by lazy evaluation y will only be evaluated if it is needed (that is, if and only if the first guard fails).

5.3 Euclid's algorithm

Consider the problem of finding the greatest common divisor, `gcd`, of two natural numbers:

```
gcd :: num -> num -> num
||pre:  nat(x) & nat(y)
||post: nat(z) & z|x & z|y (ie z is a common divisor)
||      &(A)n:nat(n|x & n|y -> n|z)
||           (ie any other common divisor divides it)
||      where z = (gcd x y)
```

We have introduced some notation in the pre- and post-conditions:

- `(A)` just means $\forall$, that is, 'for all', written in standard keyboard characters. $\exists$ would be `(E)`. Chapter 15 contains more detailed

descriptions of logical symbols.

- '|' means 'divides', or 'is a factor of'. (Note that it is not the same symbol as the division sign '`/`'.)

$$z|x \Leftrightarrow \exists y : nat.\ (x = z \times y)$$

- When we write '$y : nat$', we are using the predicate nat as though it were a Miranda type, though it is not. You can think of '$nat(y)$' and '$y : nat$' as meaning exactly the same, namely that y is a natural number. But the type-style notation is particularly useful with quantifiers:

$\exists y : nat.\ P$	means $\exists y.\ (nat(y) \wedge P)$
	('there is a natural number y for which P holds')
$\forall y : nat.\ P$	means $\forall y.\ (nat(y) \rightarrow P)$
	('for all natural numbers y, P holds')

Be sure to understand these, and in particular why it is that $\exists$ goes naturally with $\wedge$, and $\forall$ with $\rightarrow$. They are patterns that arise very frequently when you are translating from English into logic (see Chapter 15).

There is a small unexpected feature. You might expect the post-condition to say that any other common divisor is *less* than z, rather than dividing it: in other words that z is indeed the *greatest* common divisor. There is just a single case where this makes a difference, namely when x and y are both 0. All numbers divide 0, so amongst the common divisors of x and y there is no greatest one. The specification as given has the effect of specifying

```
gcd 0 0 = 0
```

Proposition 5.1 For any two natural numbers x and y, there is at most one z satisfying the specification for (`gcd` x y).

Proof Let $z1$ and $z2$ be two values satisfying the specification for (`gcd` x y); we must show that they are equal. All common divisors of x and y divide $z2$, so, in particular, $z1$ does. Similarly, $z2$ divides $z1$. Hence for some positive natural numbers p and q, we have $z1 = z2 \times p$, $z2 = z1 \times q$, so $z1 = z1 \times p \times q$ It follows that either $z1 = 0$, in which case also $z2 = 0$, or $p \times q = 1$, in which case $p = q = 1$. In either case, $z1 = z2$. □

Note that we have not actually proved that there is *any* value z satisfying the specification; only that there cannot be more than one. But we shall soon have an implementation showing how to find a suitable z, so then we shall know that there is exactly one possible result.

Euclid's algorithm relies on the following fact.

Proposition 5.2 Let x and y be natural numbers, $y \neq 0$. Then the common divisors of x and y are the same as those of y and (x `mod` y).

Proof For natural numbers x and y there are two fundamental properties of integer division, which in fact are enough to specify it uniquely: if $y \neq 0$ (pre-condition), then (post-condition)

$$x = y \times (x \texttt{ div } y) + (x \texttt{ mod } y)$$

$$0 \leq (x \texttt{ mod } y) < y$$

Suppose n is a common divisor of y and $(x \texttt{ mod } y)$. That is, there is a p such that $y = n \times p$ and a q such that $(x \texttt{ mod } y) = n \times q$. Then

$$x = y \times (x \texttt{ div } y) + (x \texttt{ mod } y) = n \times (p \times (x \texttt{ div } y) + q)$$

so n also divides x. Hence every common divisor of y and $(x \texttt{ mod } y)$ is also a common divisor of x and y. The converse is also true, by a similar proof. □

It follows that, provided $y \neq 0$, (`gcd` x y) must equal (`gcd` y (x `mod` y)). (EXERCISE: show this.) On the other hand, (`gcd` x 0) must be x. This is because $x \mid x$ and $x \mid 0$, and any common divisor of x and 0 obviously divides x, so x satisfies the specification for (`gcd` x 0). We can therefore write the following function definition:

```
gcd x y = x,                 if y=0
        = gcd y (x mod y),   otherwise
```

QUESTION: does this definition satisfy the specification?

Let us follow through the techniques that we discussed in Chapter 3. Let x and y be natural numbers, and let $z =$ (`gcd` x y). We must show that z has the properties given by the post-condition, and there are two cases corresponding to the two clauses in the definition:

$y = 0 : z = x$ We have already noted that this satisfies the specification.

$y \neq 0 : z =$ (`gcd` y (x `mod` y)) What we have seen shows that *provided that* z satisfies the specification for (`gcd` y (x `mod` y)), then it also satisfies the specification for (`gcd` x y), as required.

□

But how do we know that the recursive call gives the right answer? How do we know that it gives any answer at all? (Conceivably, the recursion might never bottom out.) Apparently, we are having to *assume* that `gcd` satisfies its specification in order to *prove* that it satisfies its specification.

5.4 Recursion variants

The answer is that we *are* allowed to assume it! But there is a catch. This apparently miraculous circular reasoning must be justified, and the key is to notice that the recursive call uses simpler arguments: the pair of arguments y with x `mod` y is 'simpler' than the pair x with y, in the sense that the second argument is smaller: x `mod` $y < y$.

As we go down the recursion, the second argument, always a natural number, becomes smaller and smaller, but never negative. This cannot go on for ever, so the recursion must eventually terminate. This at least proves termination, but it also justifies the *circular reasoning*. For suppose that `gcd` does not always work correctly. What might be the smallest bad y for which `gcd` x y may go wrong (for some x)? Not 0 — `gcd` x 0 always works correctly. Suppose Y is the smallest bad y, and `gcd` X Y goes wrong. Then $Y > 0$, so

`gcd` X Y = `gcd` Y (X `mod` Y)

But X `mod` Y is good (since X `mod` $Y < Y$), so the recursive call works correctly, so (we have already reasoned) `gcd` X Y does also — a contradiction.

We call the value y in `gcd` x y a *recursion variant* for our definition of `gcd`. It is a rough measure of the depth of recursion needed, and always decreases in the recursive calls.

Let us now state this as a reasoning principle:

> In proving that a recursive function satisfies its specification, you are *allowed to assume* that the recursive calls work correctly — *provided that* you can define a recursion variant for the function.

A recursion variant for a function must obey the following rules:

- It is calculated from the arguments of the function.
- It is a natural number (at least when the pre-conditions of the function hold). For instance, in `gcd` the recursion variant is y.
- It is calculated (trivially) from the function's arguments (x and y).
- It always decreases in the recursive calls. For the recursive call `gcd` y (x `mod` y), the recursion variant x `mod` y is less than y, the variant for `gcd` x y.

Though these rules may look complicated when stated in the abstract like this, the underlying intuitions are very basic. Although we did not mention this explicitly when deriving `gcd`, the driving force behind recursive definitions is usually to reduce the computation to simpler cases. If you can quantify this notion of simplicity, find an approximate numerical measure for it, then that is probably the basic idea for your recursion invariant.

Another example — multiplication without multiplying

Some processor chips can add and subtract, but do not have hardware instructions to multiply or divide. These operations have to be programmed. Here, in Miranda, is one method for doing this. It uses multiplication and integer division by 2, but these are easy in binary arithmetic.

A similar method can be used for exponentiation — computing x^n by using $x^{n\ div\ 2}$ (Exercise 5):

```
mult ::  num -> num -> num
||pre:   nat(n)
||post:  mult x n = x*n
||recursion variant = n
mult x n = 0,    if n=0
         = y,    if n>0 & n mod 2=0
         = y+x,  otherwise
              where y=2*(mult x(n div 2))
```

The recursion variant is n. The recursive call, used to calculate y, has variant n `div` 2. It is used when y is used, that is, the second and third alternatives, and in both of these we have $n > 0$ and so n `div` $2 < n$ — the variant has decreased.

Proposition 5.3 `mult` satisfies its specification

Proof There are three cases, corresponding to the three alternatives in the definition:

$n = 0$: `mult` $x\ n = 0 = x \times n$.

$n > 0, n$ **even:**
$$\begin{aligned} \texttt{mult}\ x\ n &= 2 \times (\texttt{mult}\ x(n/2)) \\ &= 2 \times x \times (n/2) \\ &= x \times n \end{aligned}$$

$n > 0, n$ **odd:**
$$\begin{aligned} \texttt{mult}\ x\ n &= 2 \times (\texttt{mult}\ x((n-1)/2)) + x \\ &= 2 \times x \times ((n-1)/2) + x \\ &= x \times (n-1) + x = x \times n \end{aligned}$$

□

More general properties of functions

The reasoning principle stated above concerned a particular property of a function, namely whether it satisfied its specification. But actually, the argument applied to any property of the function that you are interested in proving: as long as you have a recursion variant, then you can reason circularly by assuming that the property holds for recursive calls.

For example, consider the `sum` function of Section 5.1. The recursion variant in `sum` n is easy — it is just n itself. Having found a recursion variant, we can now prove the properties of sum, such as the following well-known equation:

Proposition 5.4 $\forall n.\ (\texttt{sum}\ n = \frac{1}{2}n(n+1))$

Proof In the non-recursive case, $n = 0$, this is obvious: both sides of the equation evaluate to 0. In the recursive case we have

$$\begin{aligned} \texttt{sum}\ n &= \texttt{sum}(n-1) + n \\ &= \tfrac{1}{2}(n-1)((n-1)+1) + n && \text{because we assume the equation holds for the recursive call} \\ &= \tfrac{1}{2}n(n+1) && \text{by a little algebra.} \end{aligned}$$

□

5.5 Mathematical induction

The reasoning principle given in the preceding section was really a packaged form of *mathematical induction*. There are two basic forms of induction and they are equivalent to each other (see Exercise 7): *simple induction* and *course of values* induction. Both should be familiar from school mathematics, but let us review them here. Both are used for proving properties of the *natural numbers*, that is, non-negative whole numbers, and both have the same underlying idea. You give a general method that shows how you can prove a property for the natural numbers one by one, starting at 0 and working up.

Simple induction

The ingredients of a simple induction proof are as follows:

- a predicate P or property on the natural numbers for which you wish to prove $\forall n : nat.\ P(n)$ (P holds for all natural numbers n);
- the **base case:** a proof of $P(0)$;
- the **induction step:** a proof of $\forall n : nat.\ (P(n) \to P(n+1))$, in other words a general method that shows for all natural numbers n how, if you had a proof of $P(n)$ (the *induction hypothesis*), you could prove $P(n+1)$.

Given these, you can indeed deduce $\forall n : nat.\ P(n)$. This is the *Principle of Mathematical Induction*. The separate parts can be put in the box proof format, as can be seen in Figure 5.3. If you were using ordinary 'forall-arrow-introduction', as in Chapter 17, you would produce a box proof such as that given in Figure 5.4. You could then consider two cases, $M = 0$ and $M = N + 1$ for some N, and so you end up more or less as in induction, proving $P(0)$ and $P(N+1)$. However, in induction, you have a free gift, the induction hypothesis $P(N)$, as an extra assumption. Without it, the proof would be difficult or even impossible.

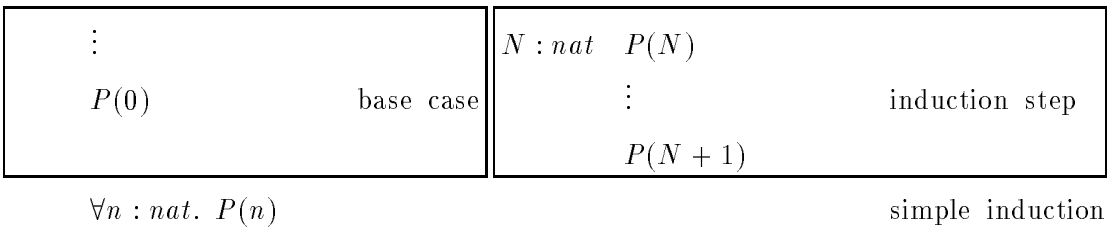

Figure 5.3 Box proof for simple induction

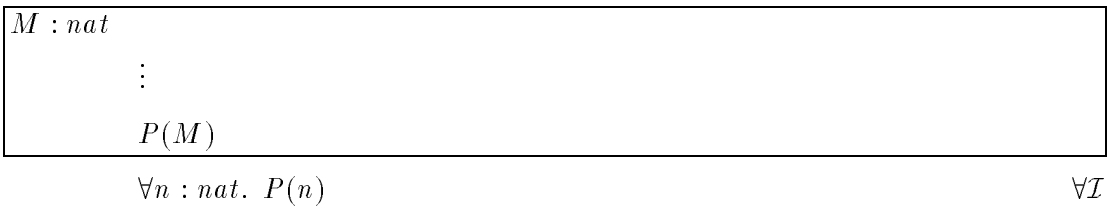

Figure 5.4

To show how this works, suppose, for instance, you want to prove $P(39976)$. The ingredients of the induction show that you can first prove $P(0)$; from this you can obtain a proof of $P(1)$; from this a proof of $P(2)$; and so on up to $P(39976)$. Of course, you never need to go through all these steps. It is sufficient to know that it can be done, and then you know that P does hold for 39976.

Another way of justifying the induction principle is by contradiction: if $\forall n : nat.\ P(n)$ is false, then there is a *smallest* n for which $P(n)$ is false. What is n? Certainly not 0, for you have proved the base case. So taking $N = n - 1$, which is still a natural number, we have $P(N)$ because n was the smallest counter-example. But now the induction step shows how to prove $P(N + 1)$, that is, $P(n)$, a contradiction. The following is a simple example.

Proposition 5.5 For all n,

$$\sum_{i=0}^{n} i^2 = \frac{n}{6}(n+1)(2n+1)$$

Proof Let $P(n)$ be the above equation, considered as a property of n. We prove $\forall n : nat.\ P(n)$ by simple induction.

base case: $n = 0$ and both sides of the equation are 0.

induction step: Suppose that P holds for N; then in the equation for $N+1$,

$$\begin{aligned} \text{LHS} &= \textstyle\sum_{i=0}^{N+1} i^2 \\ &= \textstyle\sum_{i=0}^{N} i^2 + (N+1)^2 \\ &= \tfrac{N}{6}(N+1)(2N+1) + (N+1)^2 \quad \text{by the induct. hyp.} \\ &= \tfrac{N+1}{6}(N+2)(2N+3) \\ &= \text{RHS} \end{aligned}$$

□

Course of values induction

Think of how $P(39976)$ was to be proved under simple induction: you work up to $P(39975)$, and then use the induction step. But in working up to $P(39975)$, you actually proved P for *all* natural numbers less than 39976, and it might be helpful in the induction step to use this additional information. This idea leads to a revised, *course of values* induction step (with n playing the role of what before was $n+1$):

> a general proof that shows how, if you already know that P holds for all $m < n$, you can show that P also holds for n. In logical notation,
>
> $$\forall n : nat.\ (\forall m : nat.\ (m < n \rightarrow P(m)) \rightarrow P(n))$$

Curiously enough, this also replaces the base case. When you put $n = 0$, the induction step says *if* you know $P(m)$ for all $m < 0$, then you can deduce $P(0)$; but there are no $m < 0$ (remember that we are dealing with natural numbers), so of course you know $P(m)$ for all $m < 0$. When proving the induction step, the effect is that for $n = 0$ there is no special assumption that can be used and $P(0)$ has to be proved just as before.

The *Principle of Course of Values Induction* says that if you prove the course of values induction step, then you can deduce $\forall n : nat.\ P(n)$. In box proof form, a course of values induction proof has the form seen in Figure 5.5. The following is an example.

Proposition 5.6 Every positive natural number is a product of primes. (Recall that n is prime iff it cannot be written as $p \times q$ unless either $p = 1, q = n$, or the other way round.)

Proof Let $P(n)$ be the property 'n is a product of primes' for positive natural numbers n.

Let n be a positive natural number, and suppose (course of values induction hypothesis) that every $m < n$ is a product of primes. We show that n is, too.

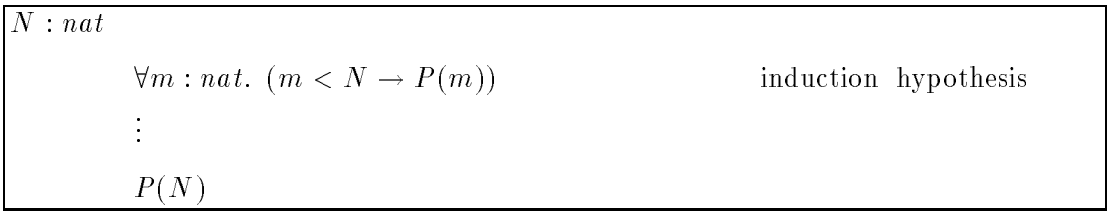

Figure 5.5

If n is itself prime, then we are done. (This also deals with the special case $n = 1$ for which there are no positive natural numbers $< n$.) If n is not prime, then we can write $n = p \times q$ for some natural numbers p and q, neither of them equal to 1. Then p and q are both less than n, so by induction each is a product of primes. Hence n is, too. □

We have actually cheated here in order to illustrate the technique in an uncomplicated way. The proof does not illustrate course of values induction on the natural numbers, but a similar principle on the *positive* natural numbers. The correct proof proves the property $P(n)$ defined by

$$P(n) \stackrel{\text{def}}{=} (n > 0 \rightarrow n \text{ is a product of primes})$$

Then there are two cases. If $n = 0$, then $P(n)$ is trivially true ('false $\rightarrow$ anything' is always true). Otherwise, $n > 0$, when we use the proof as given. When we reach $n = p \times q$, p and q must both be positive, so that from $P(p)$ and $P(q)$ we deduce that p and q are both products of primes. □

This example shows a common feature of course of values induction. It proves P for n by *reducing to simpler cases* (p and q, both smaller than n), which we assume have already been done.

5.6 Double induction — Euclid's algorithm without division

Consider the problem of finding the greatest common divisor again but this time replace the division in Euclid's algorithm by repeated subtraction:

```
gcd x y = gcd y x,      if x<y
        = x,            if y=0
        = gcd (x-y) y,  otherwise
```

y is no longer a recursion variant, because in the third clause y does not decrease: x does instead. It is still possible to concoct a recursion variant in this case, namely,

$$r(x, y) = 2 \times (x + y), \qquad \text{if } x \geq y$$

$$= 2 \times (x + y) + 1, \quad \text{if } x < y$$

However, this is somewhat artificial. The reasoning is that our notion of simplicity is not based simply on a numerical measure, but on the idea of lexicographic order:

(x', y') is simpler than (x, y) iff
$y' < y$ or
$y' = y$ and $x' < x$

You could say that y is almost a recursion variant, certainly it never *in*creases in recursive calls (unlike x). But in the case where y remains unchanged as a variant, it must be helped by x decreasing.

There is a quite general principle of *well-founded* induction (see Appendix A) that uses this idea, but, rather than going into the generalities, here we shall show how to use a double induction.

Proposition 5.7 This definition of `gcd` satisfies the specification.

Proof We use course of values induction to prove $\forall y : nat.\ P(y)$, where

$P(y) \stackrel{\text{def}}{=} \forall x : nat.$ (`gcd` x y) terminates and satisfies its post-condition)

Therefore let us take a natural number Y, and assume that $P(y)$ holds for all $y < Y$. Having fixed our Y, we now use course of values induction again to prove $P(Y)$, that is, $\forall x : nat.\ Q(x)$, where

$Q(x) \stackrel{\text{def}}{=}$ (`gcd` x Y) terminates and satisfies the post-condition.

Therefore, let us now take a natural number X, and assume that $Q(x)$ holds for all $x < X$. We prove $Q(X)$. There are three cases, as follows, for the three alternatives in the definition of `gcd`:

$X < Y$: `gcd` X Y = `gcd` Y X. By the induction hypothesis for y, $P(X)$ holds, so (`gcd` Y X) terminates and satisfies its post-condition. But the result z in the post-condition for (`gcd` Y X) is also good for (`gcd` X Y), so that is OK.

$X \geq Y$ **and** $Y = 0$: (`gcd` X Y) terminates immediately with value X, and we have argued before that X is the greatest common divisor for X and 0.

$X \geq Y$ **and** $Y > 0$: (`gcd` X Y) = (`gcd` $(X - Y)$ Y):
$X - Y$ is a natural number less than X (because $Y > 0$), so by the induction hypothesis on x we know $Q(X - Y)$. Hence (`gcd` $(X - Y)$ Y) terminates giving the greatest common divisor for $(X - Y)$ and Y, and this is also the greatest common divisor for X and Y since X and Y have the same common divisors as do $(X - Y)$ and Y.

By induction on x, we now know $\forall x : nat.\ Q(x)$, that is, $P(Y)$. Hence by induction on y we have $\forall y : nat.\ P(y)$, as required. □

5.7 Summary

- A recursive function is a function which calls itself. Functions that require the consideration of a very large number of cases (possibly infinitely many) are typically defined as recursive functions.
- Generally, a recursive function definition has a *base case* which specifies where the recursion process should end.
- When you write a recursive definition, also define a *recursion variant* for it.
- The existence of a recursion variant proves termination and allows you to reason inductively about the function.
- The *circular reasoning* is justified by mathematical induction.
- Simple induction in box proof form.

$\vdots$	$N : nat$ $\quad P(N)$ $\quad$ induction hypothesis
$P(0)$ $\quad$ base case	$\vdots$
	$P(N+1)$ $\quad$ induction step
$\forall n : nat.\ P(n)$	simple induction

- Course-of-values induction

$N : nat$ $\quad \forall m : nat.\ (m < N \rightarrow P(m))$	induction hypothesis
$\vdots$	
$P(N)$	induction step
$\forall n : nat.\ P(n)$	course of values induction

- You usually hide the induction by using the 'circular' reasoning principle for recursive definitions (once you obtain the recursion variant).
- Sometimes you need to make the induction explicit, for example, in double induction.
- Miranda's reduction strategy is called *lazy evaluation*. In lazy evaluation the evaluator evaluates an expression *only* if its result is needed.

5.8 Exercises

1. The *factorial* of a non-negative integer n is denoted as $n!$ and defined as:

   $$\texttt{factorial}\ n \quad \stackrel{\text{def}}{=} \quad \times(n-1) \times (n-2) \times (n-3) \ldots \times 2 \times 1$$

   0! is defined to be 1. Write a function `factorial` to define the factorial of a non-negative integer. Ignore the possibility of integer overflow.