Chapter 2

Equational Reasoning

There are many different formal methods used in computer science, and several of them will be introduced throughout this book. This chapter discusses one of the most powerful and yet simplest of the formal methods: *equational reasoning.*

You don't need to learn any fancy new mathematics to use equational reasoning; it is just like simple school algebra. Despite (or perhaps because of) this simplicity, equational reasoning is an effective tool for solving extremely complicated problems. Much of the most advanced current research on formal methods is based on equational reasoning, and we will use it often through the rest of this book.

2.1 Equations and Substitutions

An equation $x = y$ says that x and y have the same value; any time you see one of them you can replace it by the other. Such a replacement is called a *substitution.* For example, suppose we are given the following definitions, which are written as equations:

```
x = 8
y = 4
```

These equations are going to be cited as justifications for reasoning steps, so it helps to have a standard way to name equations. The notation { `x` } means "the equation that defines `x`". Now, suppose the problem is to evaluate $2 * x + x/y$. We do this by writing a chain of expressions, each one equal to the previous, beginning with the original problem and ending with the final result. Each step is justified by a reason given in braces { ... }.

```
2*x + x/y
  = 2*8 + 8/y        { x }
  = 2*8 + 8/4        { y }
  = 16 + 2          { arithmetic }
  = 18              { arithmetic }
```

Sometimes the justification is an explicit reference to an equation; thus the justification { x } means "the equation defining x allows this step to be taken". In other cases, where we don't want to give a formal justification because it is too trivial and tedious, an informal reason is given, for example { arithmetic }.

The format for writing a chain of equational reasoning is not rigidly specified. The reasoning above was written with each step on a separate line, with the justifications aligned to the right. This format is commonly used; it is compact and readable for many cases.

Sometimes the expressions or the justifications in a chain of reasoning become very long, and the format used above becomes unreadable. In such cases, a common approach is to place the = sign on a separate line, followed by the justification. Here is the same chain of reasoning written in the alternative notation:

```
2*x + x/y
  = { x }
2*8 + 8/y
  = { y }
2*8 + 8/4
  = { multiplication, division }
16 + 2
  = { addition }
18
```

This style takes twice as many lines, and it may seem harder to read, but experience has shown that it scales up better to very large cases. As we said before, equational reasoning is a practical tool used for large-scale proofs about realistic programs, and the expressions sometimes become quite big. The second style is becoming standard for research papers. In doing your own proofs, you may use whichever style you prefer, although it's important to be able to read both styles.

The example above is just elementary mathematics, but exactly the same technique can be used to reason about Haskell programs, because *equations in Haskell are true mathematical equations*—they are not assignment statements.

2.2 Equational Reasoning as Hand-execution

An important skill for all programmers is *hand execution*: taking a program and its data, and working out by hand what the result should be. Hand execution tells you what you think the values being computed should be, so you can locate errors by comparing the hand execution with the results of an actual execution by the computer.

For imperative programming languages, hand execution involves simulating the actions of the computer as it obeys the commands in the program. For a pure functional language, like Haskell, hand execution is done by equational reasoning instead. Thus it is not just a tool for theoreticians trying to prove theorems about software—equational reasoning is essential for all functional programmers.

For example, suppose we have the following script file:

```
f :: Integer -> Integer -> Integer
f x y = (2+x) * g y

g :: Integer -> Integer
g z = 8-z
```

Now the expression `f 3 4` can be evaluated (or "hand-executed") by equational reasoning.

```
f 3 4
  = (2+3) * g 4        { f }
  = (2+3) * (8-4)      { g }
  = 20                 { arithmetic }
```

There is an important point to notice about that example: sometimes you need to introduce parentheses to prevent different terms from getting mixed up. In the second line, we had two subexpressions being multiplied together (the first subexpression, 2+3, is enclosed in parentheses, whereas the second needs no parentheses as function application has the highest precedence). Now, suppose that somebody just blindly replaces `g 4` by `8-4`, resulting in `(2+3) * 8-4`. Now the `8-4` is really an expression on its own, yet the precedence conventions of mathematics would cause this to be interpreted as `((2+3)*8) - 4`. Notice that we prevented this error by introducing explicit parentheses, `(8-4)`, in the third line.

There is nothing really difficult going on here; you just have to be careful to introduce parentheses whenever there is a danger of confusion. An alternative approach would simply be to use fully parenthesized expressions, without relying on precedence rules at all. If you find that helpful, by all means do it! However, as you gain more experience you will probably prefer to omit most of the redundant parentheses, for the sake of improved readability.

Another issue you need to be careful with is variable names. Again, there is nothing difficult about names, but careless reasoning can lead to mistakes. The problem is that a set of equations may use the same name for different purposes. For example, consider the following equations:

```
  f x = x+1
  g x = x*5
```

Now the name x is used locally as the parameter name for both f and g, but there should be no confusion between them. When the function is applied to an argument, the argument is substituted for the parameter:

```
  f 8 * g 9
    = (8+1) * (9*5)    { f, g }
    = 405              { arithmetic }
```

In the second line, we used x twice, once with the value 8 and once with the value 9.

The best way to think of this is to consider every function as having its own local parameters. The parameters of f and g are completely different, they just happen both to be called x. The situation is just like real life: there may be (and actually are) several people named John Smith. The fact that the names are the same does not mean that the people are the same!

You can define names within a *scope* in Haskell, just as with other programming languages. A scope can be introduced implicitly: for example, the f and g defined above shouldn't be confused with unrelated functions defined in another chapter. A let expression or a where clause introduces a scope explicitly. Consider the following nested let expression:

```
  let a = 1
      b = 2
      f x = x + b  -- b=2
  in
    let b = 5
        c = 6
        g x = [x,a,b,c]  -- a=1, b=5, c=6
```

The scope of a is the entire piece of code—it is available for use everywhere. However, the b defined in b = 2 is hidden inside the inner let because the b = 5 is visible instead. The c is in scope only inside the inner let expression; you could not use it in the right hand side of f.

Many function definitions contain several equations. When you use one of these equations to justify a step of equational reasoning, it's necessary to specify which equation is being used, not just the name of the function. For example, suppose we have the following definition:

```
f :: Bool -> Int
f True = 5
f False = 6
```

If you simply give a justification { `f` }, it may not be clear to the reader which equation you are using. (At least, with large and complicated definitions that can happen, although it should be clear here!) There are two styles for writing justifications using such definitions. One way to do it is to imagine all the equations in the definition as being numbered 1, 2, 3, and so on. Then the justification { `f.2` } means "equation 2 of the definition of `f`". Another way to do it is to specify the pattern used on the left hand side of the equation, for example { `f.False` }. Either style is perfectly acceptable, and we will use both from time to time in this book.

2.2.1 Conditionals

A conditional expression satisfies the following equations:

```
  if True  then e2 else e3  =  e2      { if True }
  if False then e2 else e3  =  e3      { if False }
```

These equations are used for reasoning about programs that contain if expressions. For example, suppose `f` is defined as follows:

```
f :: Double -> Double
f x =
  if x >= 0
    then sqrt x
    else 0
```

In order to calculate the value of an application of the function, we need to use one of the if-equations.

```
  f (-3)
    = if (-3) >= 0 then sqrt (-3) else 0   { f }
    = if False then sqrt (-3) else 0       { arithmetic }
    = 0                                    { if False }
```

This example raises an interesting issue. When you are using equational reasoning, there will always be an enormous number of valid steps that could be taken. Here is another piece of equational reasoning:

```
  f (-3)
    = if (-3) >= 0 then sqrt (-3) else 0   { f }
    = if (-3) >= 0 then ERROR else 0       { sqrt }
    = if False then ERROR else 0           { arithmetic }
    = 0                                    { if False }
```

The ERROR denotes a value representing a failed computation (because the square root of a negative number is undefined). In most imperative languages, the calculation of an ERROR will abort the program. Haskell will also throw an exception if it encounters an ERROR. However, Haskell also is smart about its equational reasoning: when evaluating `if e1 then e2 else e3`, it won't attempt to evaluate either `e2` or `e3` until it knows which one is needed. Therefore Haskell begins by evaluating the conditional; then it evaluates whichever expression is required. It is perfectly acceptable to write equations that sometimes define error conditions, as long as these error values aren't actually used by the program.

2.3 Equational Reasoning with Lists

So far, the examples we have given of equational reasoning have been quite simple. To give a more realistic impression of how equational reasoning is actually used, we will give the proofs of a few simple but important theorems. First, however, we will state a few theorems without proof; those theorems will be proved in the chapter on induction, but for now we will assume their validity and use them as justifications in some simple proofs of further theorems.

The following theorem states the obvious result that the length of a concatenation is the sum of the lengths of the pieces that were concatenated. This theorem can be proved using list induction (see Chapter 4). We can use the theorem directly to justify equational reasoning steps, and an example is given below.

Theorem 1 (length (++)). Let `xs, ys :: [a]` be arbitrary lists. Then `length (xs ++ ys) = length xs + length ys`.

The next theorem says that if you map a function over a list, the list of results is the same size as the list of arguments.

Theorem 2 (length map). Let `xs :: [a]` be an arbitrary list, and `f :: a -> b` an arbitrary function. Then `length (map f xs) = length xs`.

We also need the following result, which says that if you map a function over a concatenation of two lists, the result could also be obtained by concatenating separate maps over the original lists. This theorem is frequently used in parallel functional programming, because it shows how to decompose a large problem into two smaller ones than can be solved independently.

Theorem 3 (map (++)). Let `xs, ys :: [a]` be arbitrary lists, and let `f :: a -> b` be an arbitrary function. Then `map f (xs ++ ys) = map f xs ++ map f ys`.

To illustrate how a theorem can be proved using direct equational reasoning, we will give a proof for this theorem:

Theorem 4. For arbitrary lists `xs, ys :: [a]`, and arbitrary `f :: a -> b`, the following equation holds: `length (map f (xs ++ ys)) = length xs + length ys`.

Proof. We prove the theorem by equational reasoning, starting with the left hand side of the equation and transforming it into the right hand side.

```
length (map f (xs ++ ys))
  = length (map f xs ++ map f ys)          { map (++) }
  = length (map f xs) + length (map f ys)  { length (++) }
  = length xs + length ys                  { length map }
```

There are usually many ways to prove a theorem; here is an alternative version:

```
length (map f (xs ++ ys))
  = length (xs ++ ys)                      { length map }
  = length xs + length ys                  { length (++) }
```

Either proof is fine, and you don't need to give more than one! It is a matter of taste and opinion as to which proof is better. Sometimes a shorter proof seems preferable to a longer one, but issues of clarity and elegance may also affect your choice. □

2.4 The Role of the Language

By this time, you may be wondering why you haven't been using equational reasoning for years, as it is so simple and powerful.

The reason is that equational reasoning requires *equations*, which say that two things are *equal*. Imperative programming languages simply don't contain any equations. Consider the following code, which might appear in a C or Java program:

```
n = n+1
```

That might look like an equation, but it isn't! It is an *assignment*. The meaning of the statement is a command: "fetch the value of n, then add 1 to it, then store it in the memory location called n, discarding the previous value". This is not a claim that n has the same value as n+1. Because of this, many imperative programming languages use a distinctive notation for assignments. For example, Algol and its descendants use the := operator:

```
n := n+1
```

What is different about pure functional programming languages like Haskell is that x = y *really is an equation*, stating that x and y have the same value. Furthermore, Haskell has a property called *referential transparency*. This means that if you have an equation x = y, you can replace any occurrence of x by y, and you can replace any occurrence of y by x. (This is, of course, subject to various restrictions—you have to be careful about names and scopes, and grouping with parentheses, etc.)

The imperative assignment statement `n := n+1` increases the value of `n`, whereas the Haskell equation `n = n+1` defines `n` to be the solution to the mathematical equation. This equation has no solution, so the result is that `n` is undefined. If the program ever uses an undefined value like `n`, the computer will either give an error message or go into an infinite loop.

Assignments have to be understood in the context of time passing as a program executes. To understand the `n := n+1` assignment, you need to talk about the old value of the variable, and its new value. In contrast, equations are timeless, as there is no notion of changing the value of a variable.

Indeed, there is something peculiar about the word "variable". In mathematics and also in functional programming, a variable does not vary at all! To vary means to change over time, and this simply does not happen. A mathematical equation like $x = 2^3$ means that x has the value 8, and it will still have that value tomorrow. If the next chapter says $x = 2 + 2$, then this is a *completely different variable* being defined, which just happens to have the same name; it doesn't mean that 8 shrank over the course of a few pages. A variable in mathematics means "a name that stands for some value", while a variable in an imperative programming language means "a memory location whose value is modified by certain instructions".

As one would expect, there is a price to be paid for the enormous benefits of equational reasoning. Pure functional languages like Haskell, as well as mathematics itself, are demanding in that they require you to think through a problem deeply in order to express its solution with equations. It's always possible to hack an imperative program by sticking an assignment in it somewhere in order to patch up a problem, but you cannot do that in a functional program. On the other hand, if our goal is to build correct and reliable software—and this should be our goal!—then the discipline of careful thought will be repaid in higher quality software.

2.5 Rigor and Formality in Proofs

A computer program written in a programming language like Haskell is either syntactically correct, or it is incorrect. Any violation of the syntax of the language—no matter how trivial it may seem—renders the program invalid. The reason we are so picky about the rules of the language is that software (compilers, etc.) will process programs, and software is unable to cope with the ambiguities introduced by sloppy input.

Indeed, there were attempts in the 1970s to make compilers that could figure out what the programmer meant when syntax errors were present. The compiler would go ahead and translate what the programmer obviously intended into machine language, saving the programmer from the bother of getting the program right. This worked in many cases, and yet the whole approach has fallen out of favor. The difficulty is that occasionally the compiler would get confused and the "obvious" meaning of the program was not at all what the programmer meant. The program would then go ahead and execute, producing bugs that *could not possibly* be found by comparing the output with the program text. And the more intelligent and accurate the compiler becomes, the worse the final result: when translation errors are rare, the programmer is less likely to find the ones that do occur.

Are mathematical proofs as precisely defined as computer programs? It is tempting to say that they are even more precise and picky, since they are claiming to establish fundamental mathematical truths. The reality, however, is more subtle, for two reasons: proofs may be intended for humans or for computers to read, or both; and proofs may be small (where all details can be included) or enormous (where most details must be omitted).

Consider the following equational reasoning, which was given earlier in the chapter:

```
  ...
  = (8+1) * (9*5)    { f, g }
  = 405              { arithmetic }
```

Although this is highly detailed, a number of details are left to the reader. The step taken here requires an addition and two multiplications, yet it was taken in one step rather than three, and the vague justification { `arithmetic` } was given. This proof should be convincing to a human, but it is not completely formal. We never even proved that 8+1=9.

Mathematicians make a distinction between *rigorous proofs* and *formal proofs*.

A rigorous proof is thought through clearly and carefully, and does not contain sloppy shortcuts, but it may omit trivial details that the readers should be able to work out for themselves. The point about a rigorous proof is that it includes the essential details, and skips the inessential ones. Normally, professional mathematicians will agree about the steps that can be omitted. Nonetheless, it has happened on occasion that a mathematical proof has been published and generally accepted to be valid, only for an error to be discovered years later.

A formal proof consists of solid reasoning based on a clearly specified set of axioms. Since no details are omitted and no sloppiness is allowed, computer software can be used to check the validity of a formal proof.

The degree of rigor or formality that is needed in a proof depends on the purpose for which it is intended. It is unrealistic to say that all proofs should be formal, and inadequate to say that rigor is always adequate.

Sometimes it is reasonable to give an equational reasoning proof in a *semi-formal* style, where details are omitted in order to shorten the reasoning. It is common to perform several substitutions in one step (as in the example above). Sometimes a generic justification like "arithmetic" or "calculation" is appropriate.

Strictly speaking, justifications should always be given in equational reasoning. In some cases, however, the justifications may be omitted entirely.

Sometimes this is done where the justifications should be obvious and don't really make the proof more convincing. Sometimes the justifications are omitted because they would require a lot of additional machinery (definitions, theorems, proofs) and the author feels that this would detract from the main point.

Many more examples of equational reasoning appear in this book. An excellent textbook with a focus on functional programming with equational reasoning is by Bird [4]. Equational reasoning is central to modern programming language theory, and you can see it in action in many papers appearing in *Journal of Functional Programming* and other leading publications in theoretical computer science.