Chapter 12

A simple equational calculator

This final chapter is devoted to a single programming project, the design and implementation of a simple calculator for carrying out point-free equational proofs. Although the calculator provides only a small subset of the facilities one might want in an automatic proof assistant, and is highly restrictive in a number of other ways, it will nevertheless be powerful enough to prove many of the point-free laws described in previous chapters – well, provided we are prepared to give it a nudge in the right direction if necessary. The project is also a case study in the use of modules. Each component of the calculator, its associated types and functions, is defined in an appropriate module and linked to other modules through explicit import and export lists.

12.1 Basic considerations

The basic idea is to construct a single function `calculate` with type

```
calculate :: [Law] -> Expr -> Calculation
```

The first argument of `calculate` is a list of laws that may be applied. Each law consists of a descriptive name and an equation. The second argument is an expression and the result is a calculation. A calculation consists of a starting expression and a sequence of steps. Each step consists of the name of a law and the expression that results by applying the left-hand side of the law to the current expression. The calculation ends when no more laws can be applied, and the final expression is the conclusion. The entire process is automatic, requiring no intervention on the part of the user.

Laws, expressions and calculations are each elements of appropriate data types to be defined in the following sections. But for now let us plunge straight in with an example to show the framework we have in mind.

Here are some laws (we use a smaller font to avoid breaking lines):

```
definition filter:  filter p = concat . map (box p)
definition box:     box p = if p one nil

if after dot:       if p f g . h = if (p . h) (f . h) (g . h)
dot after if:       h . if p f g = if p (h . f) (h . g)

nil constant:       nil . f = nil
map after nil:      map f . nil = nil
map after one:      map f . one = one . f

map after concat:   map f . concat = concat . map (map f)

map functor:        map f . map g = map (f . g)
map functor:        map id = id
```

Each law consists of a name and an equation. The name of the law is terminated by a colon sign, and an equation consists of two expressions separated by an equals sign. Each expression describes a function; our calculator will be one that simplifies functional expressions only (yes, it's a pointless calculator). Expressions are built from constants, like `one` and `map`, and variables, like `f` and `g`. The precise syntax will be given in due course. Note that there are no conditional laws, equations that are valid only if some subsidiary conditions are met. That will limit what we can do with the calculator, but it still leaves enough to be interesting.

Suppose we want to simplify the expression `filter p . map f`. Here is one possible calculation:

```
  filter p . map f
=   {definition filter}
  concat . map (box p) . map f
=   {map functor}
  concat . map (box p . f)
=   {definition box}
  concat . map (if p one nil . f)
=   {if after dot}
  concat . map (if (p . f) (one . f) (nil . f))
=   {nil constant}
  concat . map (if (p . f) (one . f) nil)
```

The steps of the calculation are displayed in the conventional format with the name of the law being invoked printed in braces between the two expressions to which

it applies. No more laws apply to the final expression, so that is the result of the calculation. It is certainly not simpler than the expression we started out with.

The calculator could have applied some of the laws in a different order; for example, the definition of `box` could have been applied at the second step rather than at the third. But the conclusion would have been the same. It is also possible, though not with this particular set of laws, that an expression could be simplified to different conclusions by different calculations. However, at the outset we make the decision that `calculate` returns just one calculation, not a tree of possible calculations.

Notice what is happening at each step. Some left-hand side of some law is *matched* against some subexpression of the current expression. If a match is successful the result is a *substitution* for the variables occurring in the law. For example, in the second step, the subexpression `map (box p) . map f` is successfully matched with the first map functor law, resulting in a substitution in which the variable `f` of the functor law is bound to the expression `box p`, and the variable `g` is bound to `f`. The result of the step involves *rewriting* the subexpression with the corresponding instance of the right-hand side of the law in which each variable is replaced by its binding expression. Matching, substitutions and rewriting are all fundamental components of the calculator.

Now suppose that with the same set of laws as above we want to simplify the expression `map f . filter (p . f)`. Here is the calculation:

```
    map f . filter (p . f)
=     {definition filter}
    map f . concat . map (box (p . f))
=     {map after concat}
    concat . map (map f) . map (box (p . f))
=     {map functor}
    concat . map (map f . box (p . f))
=     {definition box}
    concat . map (map f . if (p . f) one nil)
=     {dot after if}
    concat . map (if (p . f) (map f . one) (map f . nil))
=     {map after nil}
    concat . map (if (p . f) (map f . one) nil)
=     {map after one}
    concat . map (if (p . f) (one . f) nil)
```

Again, some of the laws could have been applied in a different order. No more laws apply to the final expression so that is the result of the calculation.

The point about these two calculations is that the two final expressions are the same, so we have proved

```
filter p . map f = map f . filter (p . f)
```

This is the way we will conduct equational proofs, simplifying both sides to the same conclusion. Rather than show two calculations, one after the other, the two results can be *pasted* together by recording the first calculation and then appending the steps of the second calculation in reverse. The main advantage of this scheme is simplicity; we do not have to invent a new format for proofs, and we do not have to apply laws from right to left in order to reach the desired goal. Accordingly, we will also define a function

```
prove :: [Law] -> Equation -> Calculation
```

for proving equations.

Further considerations

It is a basic constraint of our calculator that laws are applied in one direction only, namely from left to right. This is primarily to prevent calculations from looping. If laws could be applied in both directions, then the calculator could oscillate by applying a law in one direction and then immediately applying it in the reverse direction.

Even with a left-to-right rule, some laws can lead to infinite calculations. Typically, these laws are the definitions of recursive functions. For example, consider the definition of `iterate`:

```
defn iterate: iterate f = cons . fork id (iterate f . f)
```

This is the definition of `iterate` expressed in point-free form. The functions `cons` and `fork` are defined by

```
cons (x,xs) = x:xs
fork f g x  = (f x,g x)
```

We have met `fork` before in the exercises in Chapters 4 and 6, except that we wrote `fork (f,g)` instead of `fork f g`. In what follows, all our functions will be curried. The appearance of the term `iterate f` on both sides of the law means that any calculation that can apply the definition of `iterate` once can, potentially, apply it infinitely often. But not necessarily. Here is a calculation (produced by the calculator) that avoids infinite regress:

```
  head . iterate f
=   {defn iterate}
  head . cons . fork id (iterate f . f)
```

```
=   {head after cons}
  fst . fork id (iterate f . f)
=   {fst after fork}
  id
```

The calculation makes use of the two laws:

```
head after cons:  head . cons = fst
fst after fork:   first . fork f g = f
```

The reason non-termination is avoided is that these two laws are given preference over definitions in calculations, a wrinkle that we will elaborate on below.

In order to appreciate just what the calculator can and cannot do, here is another example of rendering a recursive definition into point-free form. Consider the definition of concatenation:

```
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

We will use `cat` to stand for `(++)`. We will also need `nil`, `cons` and the function `cross (f,g)`, which we will now write as `f * g`. Thus,

```
(f * g) (x,y) = (f x, g y)
```

Finally we will need a combinator `assocr` (short for 'associate-right'), defined by

```
assocr ((x,y),z) = (x,(y,z))
```

Here are the translations of the two defining equations of `cat` in point-free form:

```
cat . (nil * id)  = snd
cat . (cons * id) = cons . (id * cat) . assocr
```

We cannot prove that `cat` is associative with our calculator, for that would involve a proof by induction, but we can state it as a law:

```
cat associative: cat . (cat * id) = cat . (id * cat) . assocr
```

Continuing with this example for a bit longer, here are the two bifunctor laws of `(*)`:

```
bifunctor *:     id * id = id
bifunctor *:     (f * g) . (h * k) = (f . h) * (g . k)
```

And here is a law about `assocr`:

```
assocr law:  assocr . ((f * g) * h) = (f * (g * h)) . assocr
```

Now for the point of the example: our calculator *cannot* perform the following valid calculation:

```
  cat . ((cat . (f * g)) * h)
=   {identity law, in backwards direction}
  cat . ((cat . (f * g)) * (id . h))
=   {bifunctor *, in backwards direction}
  cat . (cat * id) . ((f * g) * h)
=   {cat associative}
  cat . (id * cat) . assocr . ((f * g) * h)
=   {assoc law}
  cat . (id * cat) . (f * (g * h)) . assocr
=   {bifunctor *}
  cat . ((id . f) * (cat . (g * h))) . assocr
=   {identity law}
  cat . (f * (cat . (g * h))) . assocr
```

The problem here is that we have to apply the identity and bifunctor laws in *both* directions, and the calculator is simply not up to the task. Observe that the essence of the proof is the simplification of the expression

```
cat . (id * cat) . assocr . ((f * g) * h)
```

in two different ways, one by using the associativity of `cat`, written in the form

```
cat associative: cat . (id * cat) . assocr = cat . (cat * id)
```

and one by using the `assocr` law. Even if we generalised `calculate` to return a tree of possible calculations, it would not be obvious what expression we would have to start out with in order to achieve the calculation above, so we abandon any attempt to get the calculator to produce it.

It is not just the functor laws that sometimes have to be applied in both directions. For an example, see Section 12.8. Sometimes we can get around the problem by stating a law in a more general form than necessary, sometimes by using a hack, and sometimes not at all. As we said at the outset, our calculator is a limited one.

In the scheme of automatic calculation that we are envisaging there are only two degrees of freedom: the choice of which law to apply, and the choice of which subexpression to be changed. The first degree of freedom can be embodied in the order in which laws are presented to the calculator: if two different laws are applicable, then the one earlier in the list is chosen.

Certainly some laws should be tried before others; these are laws that reduce the complexity of intermediate expressions. Good examples are the laws `f.id = f` and `id.f = f`. The naive definition of complexity is that there are fewer compositions on the right than on the left. It is unlikely to be a mistake to apply these

laws as soon as the opportunity arises. Indeed the fact that `id` is the identity element of composition can and will be built into the calculator, so the two identity laws will be taken care of automatically. Similarly, early application of laws like `nil.f = nil` and `map f.nil = nil` (and indeed the two laws used in the calculation about `iterate`), all of which reduce the number of compositions, help to reduce the sizes of intermediate expressions. For the sake of a word, let us call these the *simple* laws.

On the other hand, some laws should be applied only as a last resort. Typically, these laws are definitions, such as the definition of `filter` or `iterate`. For example, in the expression

```
map f . concat . map (filter p)
```

we really don't want to apply the definition of `filter` too early; rather we would prefer to apply the `map` after `concat` law first, and only apply the definition of `filter` later on if and when it becomes necessary. Apart from anything else, intermediate expressions will be shorter.

In summary it looks sensible to sort our laws into the simple laws, followed the non-simple laws that are not definitions, followed by the definitions.

The second degree of freedom is represented by the order in which the subexpressions of a given expression are presented as candidates for instances of laws: if laws are applicable to two different subexpressions, then the subexpression coming earlier in the enumeration is chosen.

That still leaves open the decision whether to give preference to laws or to subexpressions in calculations. Do we start with a subexpression and try every law in turn, or start with a law and see if it applies anywhere? Does it really matter which of these alternatives is chosen? While it is true that, having applied some law at some subexpression, the next law to be applied is likely to be at a 'nearby' expression, it is not clear how to formalise this notion of nearness, nor is it clear whether it would contribute significantly to the efficiency of calculations, either in the computation time or in the length of the result.

12.2 Expressions

At the heart of the calculator is the data type `Expr` of expressions. Most of the components of the calculator are concerned with analysing and manipulating expressions in one way or the other. Expressions are built from (function) variables

and constants, using functional composition as the basic combining form. Variables take no arguments, but constants can take any number of arguments, which are themselves expressions. We will suppose all functions are curried and there are no tuples; for example we write `pair f g` instead of `pair (f,g)`. There is no particular reason for avoiding tuples, it is just that most functions we have discussed in the book are curried and we don't really need both.

To compensate, we will also allow ourselves binary infix operators, writing, for example, `f * g` instead of `cross f g`. Except for functional composition we will not assume any order of precedence or association between binary operators, insisting that expressions involving such operators be fully parenthesised. That still leaves open the question of the precedence of composition. Does `f * g . h` mean `(f * g) . h` or `f * (g . h)`? Haskell puts composition at a high level of precedence and we will adopt the same convention. Thus `f * g . h` will be parsed as `f * (g . h)`. But we will always write such expressions using parentheses to avoid ambiguity.

Here is the proposed BNF grammar for expressions:

```
expr   ::= simple {op simple}
simple ::= term {'.' term}*
term   ::= var | con {arg}* | '(' expr ')'
arg    ::= var | con | '(' expr ')'
var    ::= letter {digit}
con    ::= letter letter {letter | digit}*
op     ::= {symbol}+
```

Variable names consist of single letters only, possibly followed by a single digit. Thus `f` and `f1` are legitimate variable names. Constant names are sequences of at least two alphanumeric characters beginning with two letters, such as `map` or `lhs2tex`, while operator names are nonempty sequences of non-alphanumeric symbols, such as `*` and `<+>`. The first line says that an expression is a simple expression, possibly followed by an operator and another simple expression. Simple expressions are compositions of terms. The remaining lines are, we trust, self-explanatory.

Here is the definition of `Expr` we will use:

```
newtype Expr = Compose [Atom] deriving Eq
data Atom    = Var VarName | Con ConName [Expr]
               deriving Eq
type VarName = String
type ConName = String
```

Expressions and atoms are declared to be members of the class `Eq` because we will need to test expressions for equality. Later on we will install expressions as an instance of `Show` for printing them at the terminal.

Here are some examples of expressions and their representations:

```
f . g . h   => Compose [Var "f",Var "g",Var "h"]
id          => Compose []
fst         => Compose [Con "fst" []]
fst . f     => Compose [Con "fst" [],Var "f"]
(f * g) . h => Compose [Con "*" [Var "f",Var "g"],Var "h"]
f * g . h   => Compose [Con "*" [Compose [Var "f"],
                                 Compose [Var "g",Var "h"]]]
```

The fact that composition is an associative operation is built into the design of `Expr`. The particular constant `id` is reserved and will always be interpreted as the identity element of composition.

The parsing combinators described in the previous chapter enable us to parse expressions. Following the BNF, we start with

```
expr :: Parser Expr
expr = simple >>= rest
 where
 rest s1 = do {op <- operator;
               s2 <- simple;
               return (Compose [Con op [s1,s2]])}
           <|> return s1
```

An operator is a sequence of one or more operator symbols, as long as it is neither the composition operator nor an equals sign:

```
operator :: Parser String
operator = do {op <- token (some (sat symbolic));
               Parsing.guard (op /= "." && op /= "=");
               return op}

symbolic = (`elem` opsymbols)
opsymbols = "!@#$%&*+./<=>?\\^|:-~"
```

The function `Parsing.guard` is an example of a *qualified* name. The Haskell Prelude also provides a function `guard`, but we want the function of the same name from a module `Parsing` that includes all our parsing functions. A qualified name consists of a module name followed by a period followed by the name of the qualified value.

A simple expression is a sequence of one or more terms separated by composition:

```
simple :: Parser Expr
simple = do {es <- somewith (symbol ".") term;
             return (Compose (concatMap deCompose es))}
```

The function `concatMap f` as an alternative to `concat . map f` is provided in the standard prelude, and `deCompose` is defined by

```
deCompose :: Expr -> [Atom]
deCompose (Compose as) = as
```

Next, a term is an identifier, either a variable or a constant, possibly with arguments, or a parenthesised expression:

```
term :: Parser Expr
term = ident args <|> paren expr
args = many (ident none <|> paren expr)
```

The parser `ident` takes a parser for a list of expressions and returns a parser for expressions:

```
ident :: Parser [Expr] -> Parser Exp
ident args
 = do {x <- token (some (sat isAlphaNum));
       Parsing.guard (isAlpha (head x));
       if isVar x
       then return (Compose [Var x])
       else if (x == "id")
       then return (Compose [])
       else
       do {as <- args;
           return (Compose [Con x as])}}
```

The test for being a variable is implemented by

```
isVar [x]   = True
isVar [x,d] = isDigit d
isVar _     = False
```

Note that any identifier consisting entirely of alphanumeric characters and beginning with a letter and which is not a variable is a constant.

Next, we make `Expr` and `Atom` instances of `Show`. As in the previous chapter we

will do this by defining `showsPrec p` for each type. A little thought reveals that we need three values for `p`:

- At top level, there is no need for parentheses. For example, we write all of `map f . map g`, `foo * baz`, and `bar bie doll` without parentheses. We assign `p=0` to this case.
- When an expression is a composition of terms, or an operator expression, occurring as an argument to a constant, we need to parenthesise it. For example, parentheses are necessary in the expression

  ```
  map (f . g) . foo f g . (bar * bar)
  ```

  But we don't have to parenthesise the middle term. We assign `p=1` to this case.
- Finally, `p=2` means we should parenthesise compositions of terms, operator expressions and curried functions of at least one argument, as in

  ```
  map (f . g) . foo (foldr f e) g . (bar * bar)
  ```

Here goes. We start with

```
instance Show Expr where
 showsPrec p (Compose [])  = showString "id"
 showsPrec p (Compose [a]) = showsPrec p a
 showsPrec p (Compose as)
  = showParen (p>0) (showSep " . " (showsPrec 1) as)
```

The last line makes use of the function `showSep`, defined by

```
showSep :: String -> (a -> ShowS) -> [a] -> ShowS
showSep sep f
 = compose . intersperse (showString sep) . map f
```

The utility function `compose` is defined by `compose = foldr (.) id`. The function `intersperse :: a -> [a] -> [a]` can be found in `Data.List` and intersperses its first argument between elements of its second. For example,

```
intersperse ',' "abcde" == "a,b,c,d,e"
```

The two occurrences of `showsPrec` on the right-hand sides of the second two clauses of `showsPrec` refer to the corresponding function for atoms:

```
instance Show Atom where
 showsPrec p (Var v)    = showString v
 showsPrec p (Con f []) = showString f
 showsPrec p (Con f [e1,e2])
```

```
  | isOp f = showParen (p>0) (showsPrec 1 e1 . showSpace .
                 showString f . showSpace . showsPrec 1 e2)
showsPrec p (Con f es)
 = showParen (p>1) (showString f . showSpace .
      showSep " " (showsPrec 2) es)

isOp f = all symbolic f
```

The value p=2 is needed in the final clause because we want parentheses in, for example, `foo (bar bie) doll`. Variables and nullary constants never need parentheses.

A module structure

The final step is to install these definitions, and possibly others, in a module for expressions. Such a module will include all the functions specifically related to expressions.

Creating such a module is not immediate because we do not yet know what other functions on expressions we may need in other modules, modules that deal with laws, calculations and so on. But for the moment we declare

```
module Expressions
  (Expr (Compose), Atom (Var,Con),
   VarName, ConName, deCompose, expr)
where
import Parsing
import Data.List (intersperse)
import Utilities (compose)
import Data.Char (isAlphaNum,isAlpha,isDigit)
```

The module `Expressions` has to be stored in a file `Expressions.lhs` to enable Haskell to find out where it resides. It exports the types `Expr` and `Atom` along with their constructors. It also exports the type synonyms `VarName` and `ConName`, as well as the functions `deCompose` and `expr`, all of which are likely to be needed in the module that deals with laws. Later on we might add more functions on expressions to this export list.

Next comes the imports. We import the module `Parsing` that contains the parsing functions, and also some functions from `Data.List` and `Data.Char`. We will also set up a module `Utilities` containing general utility functions. A good example

of a utility function is `compose`, defined above. It is not specific to expressions and may be needed in other places, so we put it into the utilities module.

12.3 Laws

We define laws in the following way:

```
data Law      = Law LawName Equation
type LawName  = String
type Equation = (Expr,Expr)
```

A law consists of a descriptive name and an equation. To parse a law we define:

```
law :: Parser Law
law = do {name <- upto ':';
          eqn <- equation;
          return (Law name eqn)}
```

The parsing function `upto c` returns the string up to but not including the character `c`, and then discards `c` if found. It wasn't included among the parsing functions of the previous chapter, but we will put it into the module `Parsing` to avoid breaking the parser abstraction. One definition is:

```
upto :: Char -> Parser String
upto c
 = Parser (\s ->
           let (xs,ys) = break (==c) s in
           if null ys then []
           else [(xs,tail ys)])
```

The parser `equation` is defined by

```
equation :: Parser Equation
equation = do {e1 <- expr;
               symbol "=";
               e2 <- expr;
               return (e1,e2)}
```

We probably don't need to show laws, but here is the definition anyway:

```
instance Show Law where
 showsPrec _ (Law name (e1,e2))
     = showString name .
```

```
        showString ": " .
        shows e1 .
        showString " = " .
        shows e2
```

The precedence number is not needed to define `showPrec` so it is made a don't care pattern. Recall that `shows` takes a printable value, here an expression, and returns a function of type `ShowS`, a synonym for `String -> String`.

Finally we sort the laws:

```
sortLaws :: [Law] -> [Law]
sortLaws laws = simple ++ others ++ defns
 where
 (simple,nonsimple) = partition isSimple laws
 (defns,others)     = partition isDefn nonsimple
```

This definition makes use of a `Data.List` function `partition` that partitions a list:

```
partition p xs = (filter p xs, filter (not . p) xs)
```

The various tests are defined by

```
isSimple (Law _ (Compose as1,Compose as2))
         = length as1 > length as2
isDefn (Law _ (Compose [Con f es], _))
         = all isVar es
isDefn _ = False
isVar (Compose [Var _]) = True
isVar _                 = False
```

The test `isVar` also appears in the module `Expressions` though with a different definition. There is no problem though since that function is not exported from the expressions module.

Here is the module declaration for laws:

```
module Laws
  (Law (Law), LawName, law, sortLaws,
   Equation, equation)
where
import Expressions
import Parsing
import Data.List (partition)
```

Having shown how to parse and print expressions and laws, we can now define two functions, one a version of `calculate` that consumes strings rather than laws and expressions:

```
simplify :: [String] -> String -> Calculation
simplify strings string
  = let laws = map (parse law) strings
        e = parse expr string
    in calculate laws e
```

In a similar vein we can define

```
prove :: [String] -> String -> Calculation
prove strings string
 = let laws = map (parse law) strings
       (e1,e2) = parse equation string
   in paste (calculate laws e1) (calculate laws e2)
```

These two functions can be put in a module `Main`. We put `paste` and `calculate` into a module concerned solely with calculations, and we turn to this module next.

12.4 Calculations

Calculations are defined by

```
data Calculation = Calc Expr [Step]
type Step        = (LawName,Expr)
```

Let's begin with the key definition of the calculator, that of `calculate`:

```
calculate :: [Law] -> Expr -> Calculation
calculate laws e = Calc e (manyStep rws e)
 where rws e = [(name,e')
                | Law name eqn <- sortedlaws,
                  e' <- rewrites eqn e,
                  e' /= e]
       sortedlaws = sortLaws laws
```

The function `rewrites :: Equation -> Expr -> [Expr]` returns a list of all the possible ways of rewriting an expression using a given equation, a function that will be defined in a separate module. It may be the case that an expression can be rewritten to itself (see Exercise H), but such rewrites are disallowed because they would lead to infinite calculations. The function `rws :: Expr -> [Step]`

returns a list of all the single steps, leading to new expressions, that can arise by using the laws in all possible ways. This list is defined by taking each law in turn and generating all the rewrites associated with the law. That means we give preference to laws over subexpressions in calculations, resolving one of the issues we worried about in the first section. Only experimentation will show if we have made the right decision.

The function `manyStep` uses `rws` to construct as many steps as possible:

```
manyStep :: (Expr -> [Step]) -> Expr -> [Step]
manyStep rws e
  = if null steps then []
    else step : manyStep rws (snd step)
    where steps = rws e
          step  = head steps
```

The calculation ends if `rws e` is the empty list; otherwise the head of the list is used to continue the calculation.

The remaining functions of the calculations module deal with showing and pasting calculations. We show a calculation as follows:

```
instance Show Calculation where
  showsPrec _ (Calc e steps)
    = showString "\n  " .
      shows e .
      showChar '\n' .
      compose (map showStep steps)
```

Each individual step is shown as follows:

```
showStep :: Step -> ShowS
showStep (why,e)
 = showString "=   {" .
   showString why .
   showString "}\n  " .
   shows e .
   showChar '\n'
```

In order to paste two calculations together we have to reverse the steps of a calculation. For example, the calculation

```
Calc e0 [(why1,e1),(why2,e2),(why3,e3)]
```

has to be turned into

```
Calc e3 [(why3,e2),(why2,e1),(why1,e0)]
```

In particular, the conclusion of a calculation is the first expression in the reversed calculation. Here is how to reverse a calculation:

```
reverseCalc :: Calculation -> Calculation
reverseCalc (Calc e steps)
 = foldl shunt (Calc e []) steps
   where shunt (Calc e1 steps) (why,e2)
         = Calc e2 ((why,e1):steps)
```

In order to paste two calculations together we first have to check that their conclusions are the same. If they are not, then we go ahead and paste the calculations anyway with an indication of failure:

```
  conc1
=  {... ??? ...}
  conc2
```

If the two conclusions are the same, we can be a little smarter than just stitching the calculations together. If the penultimate conclusion of one calculation also matches the penultimate conclusion of the other, then we can cut out the final steps altogether. And so on. Here, then, is how we paste two calculations:

```
paste :: Calculation -> Calculation -> Calculation
paste calc1@(Calc e1 steps1) calc2
  = if conc1 == conc2
    then Calc e1 (prune conc1 rsteps1 rsteps2)
    else Calc e1 (steps1 ++ (gap,conc2):rsteps2)
    where Calc conc1 rsteps1 = reverseCalc calc1
          Calc conc2 rsteps2 = reverseCalc calc2
          gap = "... ??? ..."
```

The function `prune` is defined by:

```
prune :: Expr -> [Step] -> [Step] -> [Step]
prune e ((_,e1):steps1) ((_,e2):steps2)
 | e1==e2 = prune e1 steps1 steps2
prune e steps1 steps2 = rsteps ++ steps2
 where Calc _ rsteps = reverseCalc (Calc e steps1)
```

Finally, here is the module declaration of `Calculations`:

```
module Calculations
  (Calculation (Calc), Step, calculate, paste)
```

```
where
import Expressions
import Laws
import Rewrites
import Utilities (compose)
```

The exports are those types and functions needed to define `simplify` and `prove` in the main module.

12.5 Rewrites

The sole purpose of the module `Rewrites` is to provide a definition of the function `rewrites` that appears in the definition of `calculate`. Recall that the expression `rewrites eqn e` returns a list of all expressions that can arise by matching some subexpression of `e` against the left-hand expression of `eqn` and replacing the subexpression with the appropriate instance of the right-hand expression of `eqn`.

The fun is in figuring out how to define `rewrites`. Suppose we construct a list of all possible subexpressions of an expression. We can match the given equation against each subexpression, get the substitutions that do the matching (of which there may be none, one or more than one; see the section on matching below) and compute the new subexpressions. But how do we replace an old subexpression with a new one in the original expression? The simple answer is that we can't, at least not without determining alongside each subexpression its *context* or *location* in the original expression. The new subexpression can then be inserted at this location.

Rather than introducing contexts explicitly, we take another approach. The idea is to burrow into an expression, applying a rewrite to some subexpression at some point, and then to build the rewritten expression as we climb back out of the burrow. We will need a utility function `anyOne` that takes a function yielding a choice of alternatives, and a list, and installs a single choice for one of the elements. The definition is

```
anyOne :: (a -> [a]) -> [a] -> [[a]]
anyOne f []     = []
anyOne f (x:xs) = [x':xs | x' <- f x] ++
                  [x:xs' | xs' <- anyOne f xs]
```

For example, if `f 1 = [-1,-2]` and `f 2 = [-3,-4]`, then

```
anyOne f [1,2] = [[-1,2],[-2,2],[1,-3],[1,-4]]
```

Either one of the choices for the first element is installed, or one of the choices for the second, but not both at the same time.

Here is our definition of `rewrites`:

```
rewrites :: Equation -> Expr -> [Expr]
rewrites eqn (Compose as) = map Compose (
   rewritesSeg eqn as ++ anyOne (rewritesA eqn) as)
rewritesA eqn (Var v) = []
rewritesA eqn (Con k es)
   = map (Con k) (anyOne (rewrites eqn) es)
```

In the first line we concatenate the rewrites for a *segment* of the current expression with the rewrites for any one of its proper subexpressions. Only constants with arguments have subexpressions. Note that the two uses of `anyOne` have different types, one taking a list of atoms, and one taking a list of expressions.

It remains to define `rewritesSeg`:

```
rewritesSeg :: Equation -> [Atom] -> [[Atom]]
rewritesSeg (e1,e2) as
   = [as1 ++ deCompose (apply sub e2) ++ as3
     | (as1,as2,as3) <- segments as,
       sub <- match (e1,Compose as2)]
```

The function `segments` splits a list into segments:

```
segments as = [(as1,as2,as3)
              | (as1,bs)  <- splits as,
                (as2,as3) <- splits bs]
```

The utility function `splits` splits a list in all possible ways:

```
splits :: [a] -> [([a],[a])]
splits []     = [([],[])]
splits (a:as) = [([],a:as)] ++
                [(a:as1,as2) | (as1,as2) <- splits as]
```

For example,

```
ghci> splits "abc"
[("","abc"),("a","bc"),("ab","c"),("abc","")]
```

The remaining functions `apply` and `match` have types

```
apply :: Subst -> Expr -> Expr
```

```
match :: (Expr,Expr) -> [Subst]
```

Each will be defined in their own modules, `Substitutions` and `Matchings`. Finally, here is the module declaration for `Rewrites`:

```
module Rewrites (rewrites)
where
import Expressions
import Laws (Equation)
import Matchings (match)
import Substitutions (apply)
import Utilities (anyOne, segments)
```

12.6 Matchings

The sole purpose of the module `Matchings` is to define the function `match`. This function takes two expressions and returns a list of substitutions under which the first expression can be transformed into the second. Matching two expressions produces no substitutions if they don't match, but possibly many if they do. Consider matching the expression `foo (f . g)` against `foo (a . b . c)`. There are four substitutions that do the trick: `f` may be bound to any of the expressions

```
id,   a,   a . b,   a . b . c
```

with four corresponding bindings for `g`. Although the calculator will select a single substitution at each step, it is important to take account of multiple substitutions in the process of obtaining the valid matchings. For example, in matching `foo (f . g) . bar g` against `foo (a . b . c) . bar c`, the subexpression `f . g` is matched against `a . b . c`, resulting in four possible substitutions. Only when `bar g` is matched against `bar c` are three of the substitutions rejected. A premature commitment to a single substitution for the first match may result in a successful match being missed.

The most straightforward way of defining `match (e1,e2)` is to first line up the atoms of `e1` with a partition of the atoms of `e2`; the first atom is associated with the first segment of the partition, the second with the second segment, and so on. The function `alignments` has type

```
alignments :: (Expr,Expr) -> [[(Atom,Expr)]]
```

and does the alignments. To define it we need a function `parts` that partitions a list into a given number of segments:

```
parts :: Int -> [a] -> [[[a]]]
parts 0 [] = [[]]
parts 0 as = []
parts n as = [bs:bss
             | (bs,cs) <- splits as,
               bss <- parts (n-1) cs]
```

The interesting clauses are the first two: there is one partition of the empty list into 0 segments, namely the empty partition, but there are no partitions of a nonempty list into 0 segments. For example,

```
ghci> parts 3 "ab"
[["","","ab"],["","a","b"],["","ab",""],
 ["a","","b"],["a","b",""],["ab","",""]]
```

Now we can define

```
alignments (Compose as,Compose bs)
 = [zip as (map Compose bss) | bss <- parts n bs]
   where n = length as
```

Having aligned each atom with a subexpression, we define `matchA` that matches atoms with expressions:

```
matchA :: (Atom,Expr) -> [Subst]
matchA (Var v,e) = [unitSub v e]
matchA (Con k1 es1,Compose [Con k2 es2])
   | k1==k2 = combine (map match (zip es1 es2))
matchA _ = []
```

Matching a variable always succeeds and results in a single substitution. Matching two constants succeeds only if the two constants are the same. In all other cases `matchA` returns an empty list of substitutions. The function `matchA` depends on `match`, which we can now define by

```
match :: (Expr,Expr) -> [Subst]
match = concatMap (combine . map matchA) . alignments
```

The final ingredient is the function `combine :: [[Subst]] -> [Subst]`. Each component list of substitutions in the argument of `combine` represents alternatives, so `combine` has to combine alternatives by selecting, in all possible ways, one substitution from each list and then unifying the result. We will return to this function in the module for substitutions. This completes the definition of `matches`. The module declaration is

```
module Matchings (match)
where
import Expressions
import Substitutions (Subst, unitSub, combine)
import Utilities (parts)
```

We place `parts` in the utilities module because it is not specific to expressions.

12.7 Substitutions

A substitution is a finite mapping associating variables with expressions. A simple representation as an association list suffices:

```
type Subst = [(VarName,Expr)]
```

The empty and unit substitutions are then defined by

```
emptySub    = []
unitSub v e = [(v,e)]
```

We can apply a substitution to an expression to get another expression by defining

```
apply :: Subst -> Expr -> Expr
apply sub (Compose as)
      = Compose (concatMap (applyA sub) as)
applyA sub (Var v)    = deCompose (binding sub v)
applyA sub (Con k es) = [Con k (map (apply sub) es)]
```

The function `binding` looks up a nonempty substitution for the binding for a variable:

```
binding :: Subst -> VarName -> Expr
binding sub v = fromJust (lookup v sub)
```

The function `lookup` is supplied in the Haskell Prelude and returns `Nothing` if no binding is found, and `Just e` if `v` is bound to `e`. The function `fromJust` is in the library `Data.Maybe` and removes the wrapper `Just`.

Next we tackle `combine`. This function has to combine alternative substitutions by selecting, in all possible ways, one substitution from each component list and then unifying each resulting list of substitutions:

```
combine = concatMap unifyAll . cp
```

The utility function cp, which we have seen many times before, computes the cartesian product of a list of lists.

The function unifyAll takes a list of substitutions and unifies them. To define it we first show how to unify two substitutions. The result of unification is either the union of the two substitutions if they are compatible, or no substitution if they are incompatible. To handle the possibility of failure, we can use the Maybe type, or simply return either an empty list or a singleton list. We choose the latter simply because in the following section we are going to calculate another version of the calculator, and it is simplest to stick with list-based functions:

```
unify :: Subst -> Subst -> [Subst]
unify sub1 sub2 = if compatible sub1 sub2
                  then [union sub1 sub2]
                  else []
```

In order to define compatible and union we will suppose that substitutions are maintained as lists in lexicographic order of variable name. Two substitutions are incompatible if they associate different expressions with one and the same variable:

```
compatible [] sub2 = True
compatible sub1 [] = True
compatible sub1@((v1,e1):sub1') sub2@((v2,e2):sub2')
   | v1<v2  = compatible sub1' sub2
   | v1==v2 = if e1==e2 then compatible sub1' sub2'
              else False
   | v1>v2  = compatible sub1 sub2'
```

The union operation is defined in a similar style:

```
union [] sub2 = sub2
union sub1 [] = sub1
union sub1@((v1,e1):sub1') sub2@((v2,e2):sub2')
  | v1<v2  = (v1,e1):union sub1' sub2
  | v1==v2 = (v1,e1):union sub1' sub2'
  | v1>v2  = (v2,e2):union sub1  sub2'
```

The function unifyAll returns either an empty list or a singleton list:

```
unifyAll :: [Subst] -> [Subst]
unifyAll = foldr f [emptySub]
   where f sub subs = concatMap (unify sub) subs
```

That completes the definitions we need. Here is the module declaration:

```
module Substitutions
  (Subst, unitSub, combine, apply)
where
import Expressions
import Utilities (cp)
import Data.Maybe (fromJust)
```

That makes nine modules in total for our calculator.

12.8 Testing the calculator

How useful is the calculator in practice? The only way to answer this question is to try it out on some examples. We are going to record just two. The first is the calculation we performed in Chapter 5 about pruning the matrix of choices in Sudoku. In effect we want to prove

```
filter (all nodups . boxs) . expand . pruneBy boxs
       = filter (all nodups . boxs) . expand
```

from the laws

```
defn pruneBy:      pruneBy f = f . map pruneRow . f
expand after boxs: expand . boxs = map boxs . expand
filter with boxs:  filter (p . boxs)
                     = map boxs . filter p . map boxs
boxs involution:   boxs . boxs = id
map functor:       map f . map g = map (f.g)
map functor:       map id = id
defn expand:       expand = cp . map cp
filter after cp:   filter (all p) . cp = cp . map (filter p)
law of pruneRow:   filter nodups . cp . pruneRow
                     = filter nodups . cp
```

Here is the calculation exactly as performed by the calculator, except that we have broken some expressions across two lines, a task that should be left to a pretty-printer. Don't bother to study it in detail, just note the important bit towards the end:

```
  filter (all nodups . boxs) . expand . pruneBy boxs
=   {filter with boxs}
  map boxs . filter (all nodups) . map boxs . expand .
  pruneBy boxs
=   {defn pruneBy}
  map boxs . filter (all nodups) . map boxs . expand .
  boxs . map pruneRow . boxs
=   {expand after boxs}
```

```
  map boxs . filter (all nodups) . map boxs . map boxs .
  expand . map pruneRow . boxs
=   {map functor}
  map boxs . filter (all nodups) . map (boxs . boxs) . expand .
  map pruneRow . boxs
=   {boxs involution}
  map boxs . filter (all nodups) . map id . expand .
  map pruneRow . boxs
=   {map functor}
  map boxs . filter (all nodups) . expand . map pruneRow . boxs
=   {defn expand}
  map boxs . filter (all nodups) . cp . map cp . map pruneRow . boxs
=   {map functor}
  map boxs . filter (all nodups) . cp . map (cp . pruneRow) . boxs
=   {filter after cp}
  map boxs . cp . map (filter nodups) . map (cp . pruneRow) . boxs
=   {map functor}
  map boxs . cp . map (filter nodups . cp . pruneRow) . boxs
=   {law of pruneRow}
  map boxs . cp . map (filter nodups . cp) . boxs
=   {... ??? ...}
  map boxs . filter (all nodups) . map boxs . cp . map cp
=   {defn expand}
  map boxs . filter (all nodups) . map boxs . expand
=   {filter with boxs}
  filter (all nodups . boxs) . expand
```

Yes, the calculation fails. The reason is not hard to spot: we need to apply the law

```
expand after boxs:    expand . boxs = map boxs . expand
```

in both directions, and the calculator simply cannot do that.

The solution is a hack. We add in the extra law

```
hack:    map boxs . cp . map cp = cp . map cp . boxs
```

which is just the `expand after boxs` law written in the opposite direction and with `expand` replaced by its definition. Then the calculator is happy, producing the conclusion

```
....
  map boxs . cp . map (filter nodups . cp) . boxs
=   {map functor}
  map boxs . cp . map (filter nodups) . map cp . boxs
=   {filter after cp}
  map boxs . filter (all nodups) . cp . map cp . boxs
=   {hack}
  map boxs . filter (all nodups) . map boxs . cp . map cp
=   {defn expand}
  map boxs . filter (all nodups) . map boxs . expand
```

```
=   {filter with boxs}
  filter (all nodups . boxs) . expand
```

In both cases the calculations were performed in a fraction of a second, so efficiency does not seem to be an issue. And, apart from the hack, the calculations pass muster, being almost exactly what a good human calculator would produce.

Improving the calculator

Our second example is more ambitious: we are going to use the calculator to derive another version of the calculator. Look again at the definition of `match`. This relies on `combine`, which in turn involves a messy appeal to the unification of two substitutions, with all the paraphernalia of having to test them for compatibility and computing the union. A better idea is to compute the union of two substitutions only when one of them is a unit substitution. Then everything becomes simpler and probably faster. And the technique which describes this optimisation? Yes, it's another example of accumulating parameters. Just as an accumulating parameter can avoid expensive uses of ++ operations, our hope is to avoid expensive `unify` operations.

First of all, here is the definition of `match` again, written with a couple of new subsidiary functions:

```
match = concatMap matchesA . alignments
matchesA = combine . map matchA
matchA (Var v,e) = [unitSub v e]
matchA (Con k1 es1,Compose [Con k2 es2])
   | k1==k2 = matches (zip es1 es2)
matchA _ = []
matches = combine . map match
```

Note the cycle of dependencies of these functions:

```
match --> matchesA --> matchA --> matches --> match
```

These four functions are generalised as follows:

```
xmatch sub    = concatMap (unify sub) . match
xmatchA sub   = concatMap (unify sub) . matchA
xmatches sub  = concatMap (unify sub) . matches
xmatchesA sub = concatMap (unify sub) . matchesA
```

The additional argument in each case is an accumulating parameter. Our aim will

be to obtain new versions of these definitions, whose cycle of dependencies is the same as the one above:

For the first calculation, we want to rewrite `match` in terms of `xmatch`, thereby linking the two groups of definitions. To save a lot of ink, we henceforth abbreviate `concatMap` to `cmap`. The three laws we need are

```
defn xmatch:     xmatch s = cmap (unify s) . match
unify of empty:  unify emptySub = one
cmap of one:     cmap one = id
```

In the first law we have to write `s` rather than `sub` (why?); the second two laws are the pointless versions of the facts that

```
unify emptySub sub = [sub]
cmap one xs = concat [[x] | x <- xs] = xs
```

The calculator is hardly stretched to give:

```
  xmatch emptySub
=  {defn xmatch}
  cmap (unify emptySub) . match
=  {unify of empty}
  cmap one . match
=  {cmap of one}
  match
```

Let us next deal with `xmatchA`. Because of the awkward pattern-matching style of definition of `matchA`, we simply record the following result of an easy (human) calculation:

```
xmatchA sub (Var v,e) = concat [unify sub (unitSub v e)]
xmatchA sub (Con k1 es1,Compose [Con k2 es2])
    | k1==k2 = xmatches sub (zip es1 es2)
xmatchA _ = []
```

If we introduce

```
extend sub v e = concat [unify sub (unitSub v e)]
```

then it is easy to derive

```
extend sub v e
 = case lookup v sub of
     Nothing -> [(v,e):sub]
     Just e' -> if e==e' then [sub]
                else []
```

No elaborate compatibility test, and no general union of two substitutions. Instead, as we promised earlier, we unify substitutions only with unit substitutions.

Having disposed of `xmatchA` we concentrate on the other three members of the quartet. Just as `xmatchA` is defined in terms of `xmatches`, so `xmatch` can be defined in terms of `xmatchesA`. Specifically, we want to prove that

```
xmatch s = cmap (xmatchesA s) . alignments
```

Here are the laws we need:

```
defn match:       match = cmap matchesA . alignments
defn xmatch:      xmatch s = cmap (unify s) . match
defn xmatchesA:   xmatchesA s = cmap (unify s) . matchesA
cmap after cmap:  cmap f . cmap g = cmap (cmap f . g)
```

The last, purely combinatorial law is new; we leave verification as an exercise. The calculator produces:

```
  xmatch s
=   {defn xmatch}
  cmap (unify s) . match
=   {defn match}
  cmap (unify s) . cmap matchesA . alignments
=   {cmap after cmap}
  cmap (cmap (unify s) . matchesA) . alignments
=   {defn xmatchesA}
  cmap (xmatchesA s) . alignments
```

So far, so good. That leaves us with the two remaining members of the quartet, `xmatches` and `xmatchesA`. In each case we want to obtain recursive definitions, ones that do not involve `unify`. The two functions are defined in a very similar way, and it is likely that any calculation about one can be adapted immediately to the other. This kind of meta-calculational thought is, of course, beyond the reaches of the calculator.

Let us concentrate on `xmatchesA`. We first make `xmatchesA` entirely pointless, removing the parameter `s` in the definition above. The revised definition is:

```
xmatchesA :: (Subst,[(Atom,Expr)]) -> Subst
xmatchesA = cup . (one * matchesA)
cup = cmap unify . cpp
```

where the combinator `cpp` is defined by

```
cpp (xs,ys) = [(x,y) | x <- xs, y <- ys]
```

Thus

```
    xmatchesA (sub,aes)
  = cup ([sub],aes)
  = concat [unify (s,ae) | s <- [sub],ae <- matchesA aes]
  = concat [unify (sub,ae) | ae <- matchesA aes]
```

Apart from the fact that `unify` is now assumed to be a non-curried function, this is a faithful rendition of the definition of `xmatchesA` in pointless form.

The new function `cup` has type `[Subst] -> [Subst] -> [Subst]`. Later on we will exploit the fact that `cup` is an associative function, something that `unify` could never be (why not?). As we saw in Chapter 7 the accumulating parameter technique depends on the operation of interest being associative.

The first thing to check is that the previous calculation is still valid with the new definitions. Suppose we set up the laws

```
    defn match:      match = cmap matchesA . alignments
    defn xmatch:     xmatch = cup . (one * match)
    defn xmatchesA:  xmatchesA = cup . (one * matchesA)
```

The calculator then produces

```
  xmatch
=   {defn xmatch}
  cup . (one * match)
=   {defn match}
  cup . (one * (cmap matchesA . alignments))
=   {... ??? ...}
  cmap (cup . (one * matchesA)) . cpp . (one * alignments)
=   {defn xmatchesA}
  cmap xmatchesA . cpp . (one * alignments)
```

Ah, it doesn't go through. Inspecting the gap in the calculation, it seems we need both the bifunctor law of `*` and a claim relating `cmap` and `cup`:

```
    cross bifunctor: (f * g) . (h * k) = (f . h) * (g . k)
    cmap-cup: cmap (cup . (one * g)) . cpp = cup . (id * cmap g)
```

The calculator is then happy:

```
  xmatch
=   {defn xmatch}
  cup . (one * match)
=   {defn match}
  cup . (one * (cmap matchesA . alignments))
=   {cross bifunctor}
  cup . (id * cmap matchesA) . (one * alignments)
=   {cmap-cup}
  cmap (cup . (one * matchesA)) . cpp . (one * alignments)
=   {defn xmatchesA}
```

```
cmap xmatchesA . cpp . (one * alignments)
```

That still leaves us with the claim; apart from the fact that it works we have no reason to suppose it is true. However, we can get the calculator to prove it by using another law that is not specific to matching. We leave the proof as Exercise M. Define the additional laws

```
defn cup:      cup = cmap unify . cpp
cmap-cpp: cmap (cpp . (one * f)) . cpp = cpp . (id * cmap f)
```

The calculator then produces

```
  cmap (cup . (one * g)) . cpp
=   {defn cup}
  cmap (cmap unify . cpp . (one * g)) . cpp
=   {cmap after cmap}
  cmap unify . cmap (cpp . (one * g)) . cpp
=   {cmap-cpp}
  cmap unify . cpp . (id * cmap g)
=   {defn cup}
  cup . (id * cmap g)
```

Good. It seems that the `cmap-cup` law is valid, and it even might be useful again later on. Now let us return to the main point, which is to express `xmatchesA` recursively by two equations of the form

```
xmatchesA . (id * nil)  = ...
xmatchesA . (id * cons) = ...
```

The hope is that such a definition will not involve `unify`.

It is not at all clear what laws we need for this purpose. Instead, we will write down every law we can think of that might prove useful. The first group consists of our main definitions:

```
defn match:      match = cmap matchesA . alignments
defn matchesA:   matchesA = combine . map matchA
defn xmatch:     xmatch    = cup . (one * match)
defn xmatchesA:  xmatchesA = cup . (one * matchesA)
defn xmatchA:    xmatchA   = cup . (one * matchA)
defn combine:    combine = cmap unifyAll . cp
```

The second group are some new laws about `cmap`:

```
cmap after map:    cmap f . map g = cmap (f . g)
cmap after concat: cmap f . concat = cmap (cmap f)
cmap after nil:    cmap f . nil = nil
cmap after one:    cmap f . one = f
```

The third group are some new laws about `map`:

```
map after nil: map f . nil = nil
map after one: map f . one = one . f
map after cons: map f . cons = cons . (f * map f)
map after concat: map f . concat = concat . map (map f)
```

The fourth group concerns cup:

```
cup assoc:  cup . (id * cup) = cup . (cup * id) . assocl
cup ident:  cup . (f * (one . nil)) = f . fst
cup ident:  cup . ((one . nil) * g) = g . snd
assocl: assocl. (f * (g * h)) = ((f * g) * h) . assocl
```

Finally we add in various other definitions and laws:

```
cross bifunctor: (f * g) . (h * k) = (f . h) * (g . k)
cross bifunctor: (id * id) = id
defn cp:  cp . nil = one . nil
defn cp:  cp . cons = map cons . cpp . (id * cp)
defn unifyAll: unifyAll . nil = one . nil
defn unifyAll: unifyAll . cons = cup . (one * unifyAll)
unify after nil:  unify . (id * nil) = one . fst
```

That's a total of 30 laws (including the two `map` functor laws and three laws about `cmap` that we haven't repeated). We cross our fingers and hope:

```
  xmatchesA . (id * nil)
=   {defn xmatchesA}
  cup . (one * matchesA) . (id * nil)
=   {cross bifunctor}
  cup . (one * (matchesA . nil))
=   {defn matchesA}
  cup . (one * (combine . map matchA . nil))
=   {map after nil}
  cup . (one * (combine . nil))
=   {defn combine}
  cup . (one * (cmap unifyAll . cp . nil))
=   {defn cp}
  cup . (one * (cmap unifyAll . one . nil))
=   {cmap after one}
  cup . (one * (unifyAll . nil))
=   {defn unifyAll}
  cup . (one * (one . nil))
=   {cup ident}
  one . fst
```

That's gratifying. We have shown that `xmatchesA sub [] = [sub]`. However, the recursive case cannot be established so easily. Instead we have to guess the result and then try to prove it. Here is the desired result, first expressed in pointed form and then in pointless form:

```
xmatchesA sub (ae:aes)
```

```
    = concat [xmatchesA sub' aes | sub' <- xmatchA sub ae]

xmatchesA . (id * cons)
 = cmap xmatchesA . cpp . (xmatchA * one) . assocl
```

We can perform simplification with the right-hand side (we temporarily remove the definitions of `xmatchA` and `matchesA` from `laws2`):

```
  cmap xmatchesA . cpp . (xmatchA * one) . assocl
=   {defn xmatchesA}
  cmap (cup . (one * matchesA)) . cpp . (xmatchA * one) . assocl
=   {cmap-cup}
  cup . (id * cmap matchesA) . (xmatchA * one) . assocl
=   {cross bifunctor}
  cup . (xmatchA * (cmap matchesA . one)) . assocl
=   {cmap after one}
  cup . (xmatchA * matchesA) . assocl
```

Now we would like to show

```
xmatchesA . (id * cons)
      = cup . (xmatchA * matchesA) . assocl
```

But unfortunately the calculator can't quite make it. The gap appears here:

```
  cup . ((cup . (one * matchA)) * matchesA)
=  {... ??? ...}
  cup . (one * (cup . (matchA * matchesA))) . assocl
```

The gap is easily eliminable by hand:

```
  cup . ((cup . (one * matchA)) * matchesA)
=  {cross bifunctor (backwards)}
  cup . (cup * id) . ((one * matchA) * matchesA)
=  {cup assoc}
  cup . (id * cup) . assocl . ((one * matchA) * matchesA)
=  {assocl}
  cup . (id * cup) . (one * (matchA * matchesA)) . assocl
=  {cross bifunctor}
  cup . (one * (cup . (matchA * matchesA))) . assocl
```

Once again, the inability to apply laws in both directions is the culprit. Instead of trying to force the laws into a form that would be acceptable to the calculator, we leave it here with the comment 'A hand-finished product!'.

To round off the example, here is the program we have calculated:

```
match = xmatch emptySub
xmatch sub (e1,e2)
 = concat [xmatchesA sub aes | aes <- alignments (e1,e2)]
```

```
xmatchesA sub [] = [sub]
xmatchesA sub (ae:aes)
 = concat [xmatchesA sub' aes | sub' <- xmatchA sub ae]

xmatchA sub (Var v,e) = extend sub v e
xmatchA sub (Con k1 es1,Compose [Con k2 es2])
    | k1==k2 = xmatches sub (zip es1 es2)
xmatchA _ = []
```

The missing definition is that of `xmatches`. But exactly the same treatment for `xmatchesA` goes through for `matches`, and we end up with

```
xmatches sub [] = [sub]
xmatches sub ((e1,e2):es)
 = concat [xmatches sub' es | sub' <- xmatch sub (e1,e2)]
```

Conclusions

The positive conclusion of these two exercises is that one can indeed get the calculator to assist in the construction of formal proofs. But there remains the need for substantial human input to the process, to set up appropriate laws, to identify subsidiary claims and to control the order in which calculations are carried out. The major negative conclusion is that it is a significant failing of the calculator to be unable to apply laws in both directions. The functor laws are the major culprits, but there are others as well (see the exercises for some examples). The calculator can be improved in a number of ways, but we leave further discussion to the exercises.

There are three other aspects worth mentioning about the calculator. Firstly, the complete calculator is only about 450 lines of Haskell, and the improved version is even shorter. That alone is a testament to the expressive power of functional programming. Secondly, it does seem a viable approach to express laws as purely functional equations and to use a simple equational logic for conducting proofs. To be sure, some work has to be done to express definitions in point-free form, but once this is achieved, equational logic can be surprisingly effective.

The third aspect is that, apart from parsing, no monadic code appears in the calculator. In fact, earlier versions of the calculator did use monads, but gradually they were weeded out. One reason was that we found the code became simpler without monads, without significant loss of efficiency; another was that we wanted to set things up for the extended exercise in improving the calculator. Monads are

absolutely necessary for many applications involving interacting with the world, but they can be overused in places where a purely functional approach would be smoother.

On that note, we end.

12.9 Exercises

Exercise A

Suppose we did want `calculate` to return a tree of possible calculations. What would be a suitable tree to use?

Exercise B

Why should the laws

```
map (f . g)  = map f . map g
cmap (f . g) = cmap f . map g
```

never be used in calculations, at least if they are given in the form above?

Exercise C

Here is a calculation, as recorded by the calculator

```
  map f . map g h
=   {map functor}
  map (f . g)
```

Explain this strange and clearly nonsensical result. What simple change to the calculator would prevent the calculation from being valid?

Exercise D

On the same general theme as the previous question, one serious criticism of the calculator is that error messages are totally opaque. For example, both

```
parse law "map f . map g = map (f . g)"
parse law "map functor: map f . map g   map (f . g)"
```

cause the same cryptic error message. What is it? What would be the effect of using the law

```
strange: map f . map g = map h
```

in a calculation?

Again, what change to the calculator would prevent such a law from being acceptable?

Exercise E

The definition of `showsPrec` for atoms makes use of a fact about Haskell that we haven't needed before. And the same device is used in later calculator functions that mix a pattern-matching style with guarded equations. What is the fact?

Exercise F

Define

```
e1 = foo (f . g) . g
e2 = bar f . baz g
```

List the expressions that `rewrites (e1,e2)` produces when applied to the expression `foo (a . b . c) . c`. Which one would the calculator pick?

Exercise G

Can the calculator successfully match `foo f . foo f` with the expression

```
foo (bar g h) . foo (bar (daz a) b) ?
```

Exercise H

It was claimed in the text that it is possible to apply a perfectly valid non-trivial law that will leave some expressions unchanged. Give an example of such a law and an expression that is rewritten to itself.

Exercise I

The function `anyOne` used in the definition of `rewrites` installs a single choice, but why not use `everyOne` that installs every choice at the same time? Thus if `f 1 = [-1,-2]` and `f 2 = [-3,-4]`, then

```
everyOne f [1,2] = [[-1,-3],[-1,-4],[-2,-3],[-3,-4]]
```

Using `everyOne` instead of `anyOne` would mean that a rewrite would be applied to every possible subexpression that matches a law. Give a definition of `everyOne`.

Exercise J

How many segments of a list of length n are there? The definition of `rewritesSeg` is inefficient because the empty segment appears $n+1$ times as the middle component of the segments of a list of length n. That means matching with `id` is performed $n+1$ times instead of just once. How would you rewrite `segments` to eliminate these duplicates?

Exercise K

Prove that `cmap f . cmap g = cmap (cmap f . g)`. The laws needed are:

```
defn cmap:         cmap f = concat . map f
map functor:       map f . map g = map (f.g)
map after concat:  map f . concat = concat . map (map f)
concat twice:      concat . concat = concat . map concat
```

Exercise L

The `cmap-cpp` law is as follows:

```
cmap (cpp . (one * f)) . cpp = cpp . (id * cmap f)
```

Prove it from the laws

```
cmap after cmap:  cmap f . map g = cmap (f . g)
cmap after cpp:   cmap cpp . cpp = cpp . (concat * concat)
cross bifunctor:  (f * g) . (h * k) = (f . h) * (g . k)
map after cpp:    map (f * g) . cpp = cpp . (map f * map g)
defn cmap:        cmap f = concat . map f
concat after id:  concat . map one = id
```

Can a calculator conduct the proof?

12.10 Answers

Answer to Exercise A

We would want expressions as labels of nodes and law names as labels of edges. That gives

```
type Calculation = Tree Expr LawName
data Tree a b    = Node a [(b,Tree a b)]
```

Answer to Exercise B

They would both cause the calculator to spin off into an infinite calculation. For example,

```
  map foo
=  {map functor}
  map foo . map id
=  {map functor}
  map foo . map id . map id
```

and so on.

Answer to Exercise C

The expression `map f . map g h` is perfectly valid by the rules of syntax, but of course it shouldn't be. The evaluator does not force the restriction that each appearance of one and the same constant should possess the same number of arguments. The reason the functor law can be matched successfully against the expression is that in the definition of `matchA` the function `zip` truncates the two arguments to the second `map` to one. A better calculator should check that each constant has a fixed arity.

Answer to Exercise D

The cryptic message is 'head of empty list'. The first parse fails because the law is missing its name, and the second is missing an equals sign. Use of the strange law would cause the calculator to fall over because pattern-matching with the left-hand side would not bind `h` to any expression, causing an error when the binding for `h` is requested. The calculator should have checked that every variable on the right-hand side of a law appears somewhere on the left-hand side.

Answer to Exercise E

The code for `showsPrec` takes the form

```
showsPrec p (Con f [e1,e2])
  | isOp f   = expression1 e1 e2
showsPrec p (Con f es)
  = expression2 es
```

A more 'mathematical' style would have been to write

```
showsPrec p (Con f [e1,e2])
  | isOp f    = expression1 e1 e2
  | otherwise = expression2 [e1,e2]
```

```
showsPrec p (Con f es) = expression2 es
```

The point is this: in a given clause if a pattern does not match the argument, or if it does but the guard fails to be true, the clause is abandoned and the next clause is chosen.

Answer to Exercise F

There are two rewrites, not one:

```
bar (a . b . c) . baz id . c
bar (a . b) . baz c
```

The calculator would pick the first subexpression that matches, and that means the first rewrite is chosen. Perhaps it would be better to arrange that `rewritesSeg` is applied to longer segments before shorter ones.

Answer to Exercise G

No, not with our definition of `match`. They can be matched by binding `f` to the expression `bar (daz a) b` provided `g` is bound to `daz a` and `h` to `b`, but our definition of `match` does not perform full unification.

Answer to Exercise H

To take just one example out of many, consider the law

```
if p f g . h = if (p . h) (f . h) (g . h)
```

The left-hand side matches `if a b c` with `h` bound to `id`, and the result is again the same expression.

Answer to Exercise I

The temptation is to define

```
everyOne f = cp . map f
```

but that doesn't work if `f` returns no alternatives for some element. Instead we have to define

```
everyOne :: (a -> [a]) -> [a] -> [[a]]
everyOne f   = cp . map (possibly f)
possibly f x = if null xs then [x] else xs
               where xs = f x
```

In this version, `f` returns a nonempty list of alternatives.

Answer to Exercise J

There are $(n+1)(n+2)/2$ segments of a list of length n. The improved definition is

```
segments xs = [([],[],xs] ++
              [(as,bs,cs)
              | (as,ys) <- splits xs,
                (bs,cs) <- tail (splits ys)]
```

Answer to Exercise K

The calculator produced:

```
  cmap f . cmap g
=   {defn cmap}
  concat . map f . cmap g
=   {defn cmap}
  concat . map f . concat . map g
=   {map after concat}
  concat . concat . map (map f) . map g
=   {map functor}
  concat . concat . map (map f . g)
=   {concat after concat}
  concat . map concat . map (map f . g)
=   {map functor}
  concat . map (concat . map f . g)
=   {defn cmap}
  concat . map (cmap f . g)
=   {defn cmap}
  cmap (cmap f . g)
```

Answer to Exercise L

The human proof is:

```
  cmap (cpp . (one * g)) . cpp
=   {cmap after cmap (backwards)}
  cmap cpp . map (one * g) . cpp
=   {map after cpp}
  cmap cpp . cpp . (map one * map g)
=   {cmap after cpp}
  cpp . (concat * concat) . (map one * map g)
=   {cross bifunctor}
  cpp . ((concat . map one) * concat (map g))
=   {defn cmap (backwards)}
  cpp . ((concat . map one) * cmap g)
=   {concat after id}
  cpp . (id * cmap g)
```

No, the calculation cannot be performed automatically. The `cmap after cmap`

law cannot be installed in the backwards direction without causing the calculator to loop (see Exercise B).

12.11 Chapter notes

The calculator in this chapter is based on an undocumented theorem prover by Mike Spivey, a colleague at Oxford. Ross Paterson of City University, London, has produced a version with built-in functor laws that can be applied in both directions when necessary.

One state-of-the-art proof assistant is Coq; see `http://coq.inria.fr/`.