CSE301(Functional Programming)

Lecture 1: First-order data types and pattern-matching

Noam Zeilberger

version: September 13, 2022

A **data type** is a type defined by a collection of **constructors**, each of which can take any number of arguments of different types. Informally, the concept of data type is relevant in any situation where you have something that can be built out of different kinds of things – of course there are many such situations, which is why the concept is so important! In this lecture, we will learn how data types are defined, how to write functions over them by pattern-matching, and how to perform some simple reasoning about the behavior of these functions. We will restrict to *first-order* data types, in the sense that the constructors we consider do not themselves take functions as arguments, although they can take other data types, and they can moreover be polymorphic. Such types are also sometimes called **algebraic** data types, since they obey laws similar to the algebraic laws for sums and products.

1 First example: the booleans

Let's begin by considering one of the simplest examples of a data type, the type of booleans, which in the Haskell Prelude is defined as follows:

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$

The way to read this is that *Bool* is a data type with two constructors taking no arguments, that is:

$$\mathit{False} :: \mathit{Bool}$$
$$\mathit{True} :: \mathit{Bool}$$

The definition also guarantees that these are the *only* two ways to build a value of boolean type, and it is this guarantee that enables us to define functions over the booleans by pattern-matching against these two cases.

For example, boolean negation can be written like so:

$$\mathit{not} :: \mathit{Bool} \to \mathit{Bool}$$
$$\mathit{not}\ \mathit{False} = \mathit{True}$$
$$\mathit{not}\ \mathit{True} = \mathit{False}$$

Here we have declared *not* to be a function from booleans to booleans, and defined it by its action on the two possible input values. One easy consequence of this definition is that *not* is an involution, in the sense that

$$\mathit{not}\ (\mathit{not}\ x) = x$$

for all values $x :: bool$. To prove this, it suffices to consider the two values $x = false$ and $x = true$ and apply the definition of *not*:

$$not\ (not\ False) = not\ True = False$$
$$not\ (not\ True) = not\ False = True$$

Although this "theorem" may not look that impressive, it provides us a first example of how to use reasoning-by-cases combined with purely equational reasoning to prove something about the behavior of a functional program – eventually we will see many more examples!

Next let's consider boolean conjunction:

$$both :: Bool \to Bool \to Bool$$
$$both\ False\ False = False$$
$$both\ False\ True = False$$
$$both\ True\ False = False$$
$$both\ True\ True = True$$

The first thing to pay attention to here is the type of *both*. Arrow associates to the right by default, so the type declaration that we wrote above is the same as writing $both :: Bool \to (Bool \to Bool)$. Literally, this says that *both* is a function from booleans to functions from booleans to booleans, which logically is the same thing as a function that takes a pair of booleans to a boolean. In Haskell, functions of multiple arguments are usually expressed like this as having an iterated function type, in so-called "curried" form.[1] Otherwise, we can notice that the above definition by pattern-matching essentially expresses the truth table of boolean conjunction.

The definition of boolean conjunction in the Haskell Prelude, written (&&), is actually slightly different:

$$(\&\&) :: Bool \to Bool \to Bool$$
$$False\ \&\&\ \_ = False$$
$$True\ \&\&\ b = b$$

One inessential difference with the definition of *both* is that (&&) is defined as an infix operator – but this is just "syntactic sugar". A more significant, albeit subtle, difference between the two definitions of conjunction is that (&&) implements *short-circuit evaluation*. That's because Haskell is a so-called "call-by-need" language, and arguments to functions are only evaluated when they are needed to compute the result. In this case, once the conjunction function sees that its first argument is *False*, it doesn't even need to examine its second argument (here, indicated by matching against the "wildcard" pattern _) to determine that the result should be *False*.

Although subtle, this example helps to illustrate the important difference in functional programming between *values* and *expressions*. A value of a given data type is, by definition, built using one of its constructors. An expression of that type, on the other hand, can be defined using all of the mechanisms of the language, and in general denotes a *computation* that is meant to reduce to

[1] After the logician Haskell Curry (1900 – 1982), who employed this technique in his work on combinatory logic. The technique was actually invented earlier by Moses Schönfinkel (1888 – 1942), but the term "schönfinkeling" has not really caught on.

a value. For example, *not False* is an expression of type *Bool*, which evaluates to the value *True*. In Haskell, as in most functional languages, it is possible to write down expressions that ultimately never reduce to a value. For instance, we can write down an expression for a boolean that just loops back on itself:

$$loop :: Bool$$
$$loop = loop$$

We can then use *loop* to observe the difference between the two versions of conjunction. Consider the two expressions:

$$both\ False\ loop \quad \text{vs.} \quad False\ \&\&\ loop$$

Since *loop* never reduces to either *True* or *False*, none of the four clauses in the definition of *both* can be invoked, and so the expression on the left gets stuck in an infinite loop. On the other hand, the expression on the right immediately reduces to the value *False*, since it matches the left-hand side of the first clause in the definition of (&&).

It is worth verifying this for yourself! You can do so by loading the file `DataCode.hs` associated to these notes (available on the course webpage) into the Haskell interpreter, and typing in the above expressions. You should see something roughly like the following interaction (since the first expression doesn't terminate, we type Control-C to interrupt the computation):

```
$ ghci DataCode
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( DataCode.hs, interpreted )
Ok, one module loaded.
*Main> let loop = loop
*Main> both False loop
^CInterrupted.
*Main> False && loop
False
*Main>
```

Exercise 1.1. Define the boolean disjunction function by pattern-matching, both with and without short-circuit evaluation, and write down an expression that allows one to observe the difference in behavior between the two definitions.

2 Sums and products

Besides defining particular types like *Bool*, data type declarations also provide a general way of combining one or more types to form a new type. Two important instances of this are called **sum types** and **product types**.

In Haskell, sum types are defined as follows:

$$\textbf{data}\ Either\ a\ b = Left\ a \mid Right\ b$$

Here, *Either* is called a **type constructor**, since it takes two types a and b and defines a new type *Either a b*. The data declaration automatically introduces two (value) constructors:

$$Left :: a \to Either\ a\ b$$
$$Right :: b \to Either\ a\ b$$

which, by definition, are the only two ways of forming a value of the sum type. Thinking in set-theoretic terms, the set of values of a sum type may be considered as a disjoint or "tagged" union of two sets of values.

The fact that the values of a sum type are tagged is precisely what allows us to define functions out of them by pattern-matching, without risk of ambiguity. In general, if $f :: a \to c$ and $g :: b \to c$ are two functions taking a and b, respectively, to the same target type c, then we can define a single function taking their sum to the same target by

$$h :: Either\ a\ b \to c$$
$$h\ (Left\ x) = f\ x$$
$$h\ (Right\ y) = g\ y$$

As a simple example, here is a function that coerces either a boolean or a integer[2] to an integer:

$$asInt :: Either\ Bool\ Int \to Int$$
$$asInt\ (Left\ b) = \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0$$
$$asInt\ (Right\ n) = n$$

Here the function $f :: Bool \to Int$ is given by $f\ b = \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0$, while $g :: Int \to Int$ is the identity function $g\ n = n$. As another example, we can write a function that checks whether a value of this same sum type is in fact a boolean:

$$isBool :: Either\ Bool\ Int \to Bool$$
$$isBool\ (Left\ b) = True$$
$$isBool\ (Right\ n) = False$$

In this case, $f :: Bool \to Int$ is the constant-true function $f = b \to True$ while $g :: Int \to Int$ is the constant-false function $g = n \to False$.

Whereas sum types describe values that can take multiple possible forms, product types describe values that contain multiple components. Haskell has built-in product types, written (a, b) where a and b are types. A value of type (a, b) is a pair (u, v), where $u :: a$ and $v :: b$.[3] The Haskell Prelude also provides built-in projection functions

$$fst :: (a, b) \to a$$
$$snd :: (a, b) \to b$$

which satisfy the equations $fst\ (u, v) = u$ and $snd\ (u, v) = v$.

If Haskell didn't already have built-in product types, we could define them for ourselves as a data type, for example like so:

[2]Haskell has several different types of integers, including arbitrary-length integers (or so-called "bignums"). Here we are using the type *Int* of machine-integers.

[3] This kind of overloading of notation for type constructors and value constructors is common in Haskell. Although it can be confusing at first, it does have the advantage of reducing the number of notations you need to memorize!

```
data Both a b = Pair a b
  deriving (Show, Eq)
```

Of course, the names that we gave to the type and value constructors are arbitrary.[4] The "deriving" line is just a bit of Haskell syntax to automatically derive pretty-printing and equality-testing routines for values of our newly-defined data type, which is useful for testing. (We will talk more about *Show*, *Eq*, and other type classes in Lecture **??**.) Once again, the way to read the data type declaration itself is that it introduces a single constructor taking two arguments

```
Pair :: a → b → Both a b
```

and asserts that this is the *only* way to build a value of the product type. In other words, a value of type *Both a b* necessarily contains both a value of type *a* and a value of type *b*.

Using pattern-matching, we can write projection functions for our own version of product types (analogues of the Prelude functions *fst* and *snd*):

```
projLeft :: Both a b → a
projLeft (Pair u v) = u
projRight :: Both a b → b
projRight (Pair u v) = v
```

As a slightly more interesting example, we can write a pair of coercion functions witnessing the fact that having both A and either B or C is essentially the same thing as having either A and B or having A and C:

```
coerceTo :: Both a (Either b c) → Either (Both a b) (Both a c)
coerceTo (Pair x (Left y)) = Left (Pair x y)
coerceTo (Pair x (Right y)) = Right (Pair x y)

coerceFrom :: Either (Both a b) (Both a c) → Both a (Either b c)
coerceFrom (Left (Pair x y)) = Pair x (Left y)
coerceFrom (Right (Pair x y)) = Pair x (Right y)
```

It is not hard to check that the equations

$$coerceFrom\ (coerceTo\ u) = u \quad \text{and} \quad coerceTo\ (coerceFrom\ v) = v$$

hold for all $u :: Both\ a\ (Either\ b\ c)$ and $v :: Either\ (Both\ a\ b)\ (Both\ a\ c)$. This means that not only can we convert values of one type into values of the other, and vice versa, but moreover these transformations are reversible. Formally, we say that the pair of coercion functions realize a **type isomorphism**

$$Both\ a\ (Either\ b\ c) \cong Either\ (Both\ a\ b)\ (Both\ a\ c) \tag{1}$$

which is a more mathematically precise way of saying that two types are "essentially equivalent". This particular isomorphism (1) expresses the *distributivity* of product types over sum types, and is as an analogue of the algebraic identity

$$a(b + c) = ab + ac \tag{2}$$

[4] And if we were to follow typical Haskell naming conventions, it might be a bit more idiomatic to reuse the same name for both, see Footnote 3.

familiar from high school algebra.

Finally, both sum types and product types can be generalized from these binary versions to combine any number of types, including *no* types. The nullary product type is called the **unit type** and is written () in Haskell, with a single value also written (). Of course we can define our own isomorphic version, as a data type with a single constructor taking no arguments:

```
data Unit = U
  deriving (Show, Eq)
```

The nullary sum type is called the **zero type** (or *void type*), and corresponds to a data type with *no constructors*. We can express this in Haskell (since the 2010 revision of the language) using the following syntax:

```
data Zero
```

Exercise 2.1. For each of the following valid type isomorphisms, write a pair of coercion functions realizing said isomorphism:

$$Either\ a\ (Either\ b\ c) \cong Either\ (Either\ a\ b)\ c \tag{1}$$
$$Either\ a\ b \cong Either\ b\ a \tag{2}$$
$$Both\ a\ (Both\ b\ c) \cong Both\ (Both\ a\ b)\ c \tag{3}$$
$$Both\ a\ b \cong Both\ b\ a \tag{4}$$
$$Both\ Unit\ a \cong a \tag{5}$$
$$Either\ Zero\ a \cong a \tag{6}$$

Exercise 2.2. Write a pair of functions of the following types:

```
f :: Either a (Both b c) → Both (Either a b) (Either a c)
g :: Both (Either a b) (Either a c) → Either a (Both b c)
```

Does this show that $Either\ a\ (Both\ b\ c) \cong Both\ (Either\ a\ b)\ (Either\ a\ c)$? Explain why or why not.

3 Lists and structural induction

Lists are ubiquitous in functional programming, appearing frequently both in the interfaces to functions as well as in the intermediate results of computations. In Haskell, the list type is notated $[a]$, with the type a of the underlying elements of the list surrounded in brackets. Modulo syntax, lists are defined by the following data type declaration:

```
data [a] = [] | a : [a]
```

That is, a list of as is either the empty list, written $[]$ and pronounced "nil", or constructed by prepending an element of type a to a list of as, with this binary operation written $x : xs$ and pronounced (for historical reasons) "x cons xs". Note that lists are an example of a *recursive* data type, since they are defined in terms of themselves.

The above is not quite valid Haskell syntax, since Haskell does not allow operators like $[\_]$ to be defined by data declarations. If we want, we can define our own isomorphic version of lists via the following perfectly legal Haskell data type declaration:

$$\begin{array}{l}\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\ \quad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})\end{array}$$

The following pair of functions realize an isomorphism $[a] \cong \mathit{List}\ a$:

$$\begin{array}{l}\mathit{toList} :: [a] \to \mathit{List}\ a \\ \mathit{toList}\ [\,] = \mathit{Nil} \\ \mathit{toList}\ (x : xs) = \mathit{Cons}\ x\ (\mathit{toList}\ xs) \\ \\ \mathit{fromList} :: \mathit{List}\ a \to [a] \\ \mathit{fromList}\ \mathit{Nil} = [\,] \\ \mathit{fromList}\ (\mathit{Cons}\ x\ xs) = x : \mathit{fromList}\ xs\end{array}$$

However, let's stick to Haskell's built-in syntax for lists in the rest of these notes.

Two basic operations that one can perform on a list are to extract its head and its tail, defined as follows:

$$\begin{array}{l}\mathit{head} :: [a] \to a \\ \mathit{head}\ (x : xs) = xs \\ \\ \mathit{tail} :: [a] \to a \\ \mathit{tail}\ (x : xs) = xs\end{array}$$

Observe that these definitions are only *partial*, in the sense that they do not specify a behavior when the input is nil $[\,]$. By default, Haskell does not require pattern-matching definitions to cover all possible patterns, and it is clear that there is no good choice of values to return in this case (since the empty list does not have a head or tail). It is generally considered good practice to cover all possible patterns for the input type, though, and to explicitly raise an exception in the case of unexpected input – indeed, that's what's done for the definitions of *head* and *tail* that appear in the Haskell Prelude. In any case, the Haskell compiler will generate a pattern-matching exception by default in the case that the input does not match any one of the provided patterns, so that the behavior of the function on any input is always well-defined.

Another basic operation on lists is concatenation. The concatenation of two lists xs and ys is notated $xs +\!\!+ ys$ in Haskell, and defined as follows:

$$\begin{array}{l}(+\!\!+) :: [a] \to [a] \to [a] \\ [\,] +\!\!+ ys = ys \\ (x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)\end{array}$$

Observe that this is a *recursive* definition: the concatenation of $x : xs$ with ys is defined as the list obtained by cons'ing x to the front of the concatenation of xs with ys. In general, recursive definitions may lead to non-termination (as we saw with the example of *loop* above), but in this case the circularity is not "vicious" since the first argument to $(+\!\!+)$ gets smaller. Indeed, we can prove that $xs +\!\!+ ys$ is a well-defined list by induction on the length of xs, or alternatively by *structural induction* on xs itself.

The principle of **structural induction over lists** says that for any property $P(xs)$ of a list xs, if we can show both of the following:

1. $P([\,])$ holds
2. for any element x and list xs, $P(xs)$ implies $P(x : xs)$

then $P(xs)$ holds for all lists xs. Although it may seem like overkill for this example, let's take the time to spell out how to prove that $(+\!\!+)$ is well-defined by structural induction. To that end, we take $P(xs)$ to be the property that "for every list ys, there is a well-defined list zs such that $xs +\!\!+ ys = zs$". We need to show two things:

1. $P([\,])$: by definition $[\,] +\!\!+ ys = ys$, so we take $zs = ys$.
2. $P(xs)$ implies $P(x{:}xs)$: by the inductive assumption, there is a list zs' such that $xs +\!\!+ ys = zs'$. But by definition $(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) = x : zs'$, so we take $zs = x : zs'$.

We have thus established that $P(xs)$ holds for all lists xs, i.e., that for all xs and ys, there is a well-defined zs such that $xs +\!\!+ ys = zs$.

Exercise 3.1. Classically, a *monoid* is defined as a set equipped with an associative binary operation and an identity element. For example, the natural numbers form a monoid under addition with zero as the identity element, written $(\mathbb{N}, +, 0)$, and also a monoid under multiplication with one as the identity element, written $(\mathbb{N}, \times, 1)$. Show that lists form a monoid under concatenation with nil as the identity element, by proving that the equations:

$$[\,] +\!\!+ xs = xs \tag{1}$$
$$xs +\!\!+ [\,] = xs \tag{2}$$
$$xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs \tag{3}$$

hold for all lists xs, ys, and zs. *Hint:* Use structural induction for (2) and (3).

Exercise 3.2. A *monoid homomorphism* from $(X, *, e_X)$ to $(Y, \bullet, e_Y)$ is defined as a function $\phi : X \to Y$ that is compatible with both monoid structures, in the sense that $\phi(x_1 * x_2) = \phi(x_1) \bullet \phi(x_2)$ for all $x_1, x_2 \in X$ and moreover $\phi(e_X) = e_Y$. Prove that the length function:

$$\begin{aligned}&\mathit{length} :: [\,a\,] \to \mathit{Int}\\ &\mathit{length}\ [\,] = 0\\ &\mathit{length}\ (x : xs) = 1 + \mathit{length}\ xs\end{aligned}$$

defines a monoid homomorphism from lists under concatenation to the natural numbers under addition. *Hint:* Use structural induction, and some properties of addition.

Exercise 3.3. Let $(b, \circledast, e)$ be an arbitrary monoid represented in Haskell, i.e., a type b, a binary operation $(\circledast) : b \to b \to b$, and a value $e :: b$ such that

$$e \circledast v = v = v \circledast e \qquad u \circledast (v \circledast w) = (u \circledast v) \circledast w$$

for all values $u, v, w :: b$. Suppose moreover that you are given a function $f : a \to b$. Write a function $\phi : [\,a\,] \to b$ satisfying both of the following properties:

- ϕ behaves like f for singleton lists, i.e., $\phi\ [x] = f\ x$ for all $x :: a$, and

- ϕ is a monoid homomorphism.

Finally, explain how to recover the *length* function from Exercise 3.2 as ϕ for a particular choice of monoid $(b, \circledast, e)$ and function $f : a \to b$.

Exercise 3.4. Prove that the function ϕ you defined in Exercise 3.3 is essentially unique, in the following sense: if $\psi :: [a] \to b$ is any other monoid homomorphism such that $\psi\ [x] = f\ x$ for all $x :: a$, then $\psi\ xs = \phi\ xs$ for all $xs :: [a]$. (As a result of all these properties, we can conclude that the type of lists $[a]$ equipped with concatenation forms the *free monoid* over a.)

4 Maybe types

It sometimes happens that we want a function to try to compute something that *may* yield a value of some type, but might also fail, and we want the function to tell us explicitly when it fails. In Haskell, this is achieved by having the function return a value of the *Maybe* type, which is defined as follows:

```
data Maybe a = Nothing | Just a
```

The type *Maybe a* is called a **maybe type**, or alternatively *option type* (which is the terminology in Standard ML and OCaml). Observe that *Maybe a* $\cong$ *Either* () *a*, meaning that maybe types can be isomorphically encoded using sum types and the unit type...although maybe types are such a common use pattern that they deserve their own syntax!

To give an example application of maybe types, the Prelude defines a function *lookup* which tries to find a value matching to a key in a list of key-value pairs, otherwise returning *Nothing*:[5]

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup k []      = Nothing
lookup k ((k', v) : kvs)
  | k == k'      = Just v
  | otherwise = lookup k kvs
```

As another example, the Standard Library function *elemIndex* tries to find the index of an element within a list:

```
elemIndex :: Eq a => a -> [a] -> Maybe Int
elemIndex x [] = Nothing
elemIndex x (x' : xs)
  | x == x'      = Just 0
  | otherwise  = case elemIndex x xs of
                   Nothing -> Nothing
                   Just i -> Just (i + 1)
```

Notice that here we use pattern-matching on the result of the recursive call, via Haskell's **case** construct.

[5]This definition makes use of Haskell's *pattern guards*, which extend ordinary pattern-matching against constructors by allowing tests of arbitrary boolean conditions.

Exercise 4.1. Prove the type isomorphism

$$\mathit{Both}\ (\mathit{Maybe}\ a)\ (\mathit{Maybe}\ b) \cong \mathit{Maybe}\ (\mathit{Either}\ (\mathit{Either}\ a\ b)\ (\mathit{Both}\ a\ b))$$

which is the analogue of the algebraic identity $(1 + a)(1 + b) = 1 + a + b + ab$.

5 Introducing accumulators

Although the connections between functional programming and mathematics are fascinating, eventually one has to contend with the fact that in programming, we care not only about mathematical correctness but also about *efficiency*. There may be different ways of implementing the same function that although equivalent in terms of their input-output behavior, differ wildly in terms of their consumption of time, space, or other resources – and understanding these costs is an important part of learning functional programming.

The use of *accumulators* is a general technique that allows to derive relatively efficient functional programs, by in a sense mimicking the behavior of a corresponding imperative program. Consider the list reversal function. We can give a straightforward recursive implementation that reduces list reversal to list concatenation:

$$\begin{aligned}
&\mathit{reverse} :: [a] \to [a] \\
&\mathit{reverse}\ [\,] = [\,] \\
&\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs +\!\!+ [x]
\end{aligned}$$

It is easy to verify that this implementation of list reversal is correct, and even to establish that it satisfies expected properties such as $\mathit{reverse}\ (xs +\!\!+ ys) = \mathit{reverse}\ ys +\!\!+ \mathit{reverse}\ xs$ and $\mathit{reverse}\ (\mathit{reverse}\ xs) = xs$. However, this implementation of *reverse* is inefficient. The issue is that computing the concatenation of two lists takes time proportional to the length of the first list. Given a list xs of length n, $\mathit{reverse}\ xs$ makes n calls to concatenation of the form $xs' +\!\!+ [x]$, where xs' is the result of calling *reverse* on a sublist of xs of length $0 \leq k \leq n$. Since the *reverse* function is length-preserving, the overall time complexity of this implementation of *reverse* is $\Theta(n^2)$ in the length of the input list.

On the other hand, there is a simple imperative algorithm for reversing a list in $\Theta(n)$ time, using an auxiliary stack:

1. Initialize the stack to be empty.
2. While the input list is non-empty, push its head onto the stack, and keep processing its tail.
3. Once the input list is empty, return the contents of the stack.

Perhaps surprisingly, we can turn this imperative solution into an elegant functional program! The idea is to define a helper function which, in addition to the input list xs, takes the auxiliary "stack" (which is in fact just another list) as an extra input argument ys:

$$\begin{aligned}
&\mathit{revacc} :: [a] \to [a] \to [a] \\
&\mathit{revacc}\ [\,]\ ys = ys \\
&\mathit{revacc}\ (x : xs)\ ys = \mathit{revacc}\ xs\ (x : ys)
\end{aligned}$$

The first clause $revacc\ [\,]\ ys = ys$ of the definition corresponds to step (3) of the imperative algorithm above, while the clause $revacc\ (x{:}xs)\ ys = revacc\ xs\ (x{:}ys)$ corresponds to step (2). Finally, step (1) is implemented by (re-)defining *reverse* in terms of *revacc*:

$$reverse\ xs = revacc\ xs\ [\,]$$

We have thus obtained an efficient, purely functional implementation of list reversal. Here's an example of it in action:

$$\begin{aligned}
&reverse\ [1,2,3,4]\\
&= revacc\ [1,2,3,4]\ [\,]\\
&= revacc\ [2,3,4]\ [1]\\
&= revacc\ [3,4]\ [2,1]\\
&= revacc\ [4]\ [3,2,1]\\
&= revacc\ [\,]\ [4,3,2,1]\\
&= [4,3,2,1]
\end{aligned}$$

We can see that the second argument of *revacc* really behaves like a stack, and that the reversal is indeed computed in linear time.

The second argument of *revacc* is said to be an "accumulator", since it accumulates intermediate results on the way towards computing the final answer. That gives a very operational way of understanding the accumulator technique, but there is also a slightly more conceptual way of understanding it, related to the idea that to solve a particular problem, oftentimes it can be helpful to try to solve a *more general problem.* In this instance, the function *revacc* actually solves the following more general problem: given two lists, compute the reversal of the first list concatenated with the second list, that is, $revacc\ xs\ ys = reverse\ xs +\!\!+ ys$. After writing this more general function, we can then reduce list reversal to the case $ys = [\,]$.

Another classic example of the use of accumulators is computing Fibonacci numbers. Mathematically, the Fibonacci numbers F_n are defined by the recurrence $F_n = F_{n-1} + F_{n-2}$ together with initial values $F_0 = 0$, $F_1 = 1$, which translates directly to the following Haskell code:[6]

$$\begin{aligned}
&fib :: Integer \to Integer\\
&fib\ n\\
&\quad \mid n \equiv 0 = 0\\
&\quad \mid n \equiv 1 = 1\\
&\quad \mid n \geqslant 2 = fib\ (n-1) + fib\ (n-2)
\end{aligned}$$

Although mathematically correct, this recursive computation of the Fibonacci numbers is horribly inefficient: each call to *fib* spawns two recursive calls to *fib* while only decreasing the input by either 1 or 2, and so $fib\ n$ has complexity exponential in n. We can see this by running the program on a few inputs:

```
*Main> :set +s  -- ask ghci to print time and space usage
*Main> fib 10
55
(0.02 secs, 123,512 bytes)
```

[6]Here we use the type *Integer* of arbitrary-length integers, rather than machine integers.

```
*Main> fib 20
6765
(0.08 secs, 6,423,944 bytes)
*Main> fib 30
832040
(2.38 secs, 781,578,344 bytes)
*Main> fib 31
1346269
(3.58 secs, 1,264,577,008 bytes)
*Main> fib 32
2178309
(6.05 secs, 2,046,084,072 bytes)
```

We observe that the behaviour is indeed exponential, with each increment of n multiplying both the time and space usage by roughly the golden ratio.

On the other hand, there is a much more efficient imperative algorithm for computing F_n in linear time, using a pair of auxiliary variables a and b:

- Initialize $a \leftarrow 0$ and $b \leftarrow 1$.
- While $n > 0$, simultaneously update $(a, b) \leftarrow (b, a + b)$, and decrement n.
- Once $n = 0$, return the value of a.

Again, this imperative solution can be transformed almost mechanically into a purely functional one. First we define a helper function *fibacc* taking two extra parameters:

$$
\begin{aligned}
&\mathit{fibacc} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
&\mathit{fibacc}\ n\ a\ b \\
&\quad \mid n \equiv 0 = a \\
&\quad \mid n \geqslant 1 = \mathit{fibacc}\ (n-1)\ b\ (a+b)
\end{aligned}
$$

And then we redefine *fib* as an appropriate call to *fibacc*:

$$\mathit{fib}\ n = \mathit{fibacc}\ n\ 0\ 1$$

As we saw before with *revacc*, the function *fibacc* may be seen as solving a more general problem than computing Fibonacci numbers: *fibacc* n a b actually computes the nth entry of a *generalized Fibonacci sequence,* defined by the same recurrence but with initial values a and b. (For example, *fibacc* n 2 1 is the nth "Lucas number".)

We can observe that the new version indeed computes Fibonacci numbers much more efficiently:

```
*Main> fib n = fibacc n 0 1  -- redefine fib
*Main> fib 32
2178309
(0.00 secs, 82,288 bytes)
*Main> fib 100
354224848179261915075
(0.01 secs, 114,400 bytes)
*Main> fib 1000
```

```
434665576869374564356885276750406258025646605173717804024817290 89 \
536555417949051890403879840079255169295922593080322634775209689 62 \
323987332247116164299644090653318793829896964992851600370447613 77 \
95166849228875
(0.01 secs, 637,736 bytes)
```

6 Trees

Trees give another important example of a data type. In fact, there are many different kinds of "trees" in computer science and mathematics (not to mention in botany), but for the sake of concreteness let's consider binary trees with labelled nodes. In Haskell, these can be defined via the following recursive data type declaration:

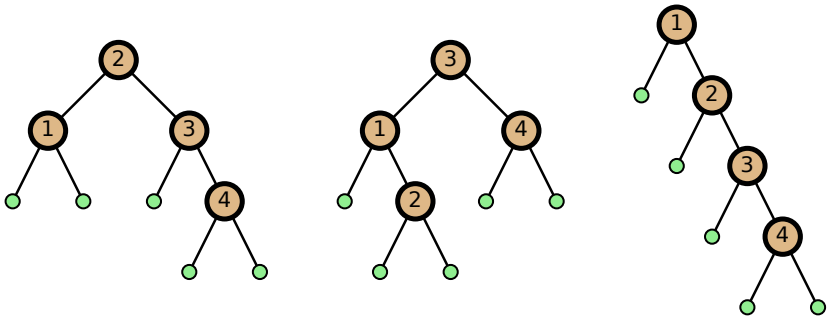
> **data** *BinTree a* = *Leaf* | *Node a* (*BinTree a*) (*BinTree a*)
> **deriving** (*Show*, *Eq*)

which, again to be clear, introduces the constructors *Leaf* and *Node* with the following types:

> *Leaf* :: *BinTree a*
> *Node* :: *a* → *BinTree a* → *BinTree a* → *BinTree a*

So a binary tree is either a *Leaf*, containing no data, or a *Node*, which contains a value together with a left subtree and a right subtree.

For example, the binary trees



can be represented as the following values, respectively:

> *t1*, *t2*, *t3* :: *BinTree Int*
> *t1* = *Node* 2 (*Node* 1 *Leaf Leaf*) (*Node* 3 *Leaf* (*Node* 4 *Leaf Leaf*))
> *t2* = *Node* 3 (*Node* 1 *Leaf* (*Node* 2 *Leaf Leaf*)) (*Node* 4 *Leaf Leaf*)
> *t3* = *Node* 1 *Leaf* (*Node* 2 *Leaf* (*Node* 3 *Leaf* (*Node* 4 *Leaf Leaf*)))

Many different operations on binary trees can be naturally defined using pattern-matching and recursion:

- *(Compute the size of a binary tree in # nodes.)*

$$nodes :: BinTree\ a \to Int$$
$$nodes\ Leaf = 0$$
$$nodes\ (Node\ \_\ tL\ tR) = 1 + nodes\ tL + nodes\ tR$$

```
*Main> (nodes t1, nodes t2, nodes t3)
(4,4,4)
```

- *(Compute the size of a binary tree in # leaves.)*

$$leaves :: BinTree\ a \to Int$$
$$leaves\ Leaf = 1$$
$$leaves\ (Node\ \_\ tL\ tR) = leaves\ tL + leaves\ tR$$

```
*Main> (leaves t1, leaves t2, leaves t3)
(5,5,5)
```

- *(Compute the height of a binary tree = maximum length path to a leaf.)*

$$height :: BinTree\ a \to Int$$
$$height\ Leaf = 0$$
$$height\ (Node\ \_\ tL\ tR) = 1 + max\ (height\ tL)\ (height\ tR)$$

```
*Main> (height t1, height t2, height t3)
(3,3,4)
```

- *(Compute the mirror image of a tree.)*

$$mirror :: BinTree\ a \to BinTree\ a$$
$$mirror\ Leaf = Leaf$$
$$mirror\ (Node\ x\ tL\ tR) = Node\ x\ (mirror\ tR)\ (mirror\ tL)$$

```
*Main> mirror t2
Node 3 (Node 4 Leaf Leaf) (Node 1 (Node 2 Leaf Leaf) Leaf)
```

Exercise 6.1. Write a function $isBST :: Ord\ a \Rightarrow BinTree\ a \to Bool$ which determines whether a binary tree is a binary search tree.