The Haskell School of Music

by

Paul Hudak

Yale University
Department of Computer Science

Contents

Preface

In 2000 I wrote a book called *The Haskell School of Expression – Learning Functional Programming through Mulitmedia.* In that book I used graphics, animation, music, and robotics as a way to motivate learning how to program, and specifically how to learn *functional programming* using Haskell, a purely functional programming language. Haskell is quite a bit different from conventional imperative or objected-oriented languages such as C, C++, Java, C#, and so on. It takes a different mind-set to program in such a language, and appeals to the mathematically inclined and to those who seek purity and elegance in their programs. Although Haskell was designed almost twenty years ago, it has only recently begun to catch on, not just because of its purity and elegance, but because with it you can solve real-world problems quickly and efficiently, and with great economy of code.

I have also had a long, informal, yet passionate interest in music, being an amateur jazz pianist and having played in several bands over the years. About ten years ago, in an effort to combine work with play, I wrote a Haskell library called *Haskore* for expressing high-level computer music concepts in a purely functional way. Indeed, three of the chapters in *The Haskell School of Expression* summarize the basic ideas of this work. Thus, when I recently became responsible for the Music Track in the new *Computing and the Arts* major at Yale, and became responsible for teaching not one, but two computer music courses in the new curriculum, it was natural to base the course material on Haskell. This current book is essentially a rewrite of *The Haskell School of Expression* with a focus entirely on music, based on, and improving upon, the ideas in Haskore.

Haskell was named after the logician Haskell B. Curry who, along with Alonzo Church, established the theoretical foundations of functional programming in the 1940's, when digital computers were mostly just a gleam in researchers' eyes. A curious historical fact is that Haskell Curry's father, Samuel Silas Curry, helped found and direct a school in Boston called the *School of Expression.* (This school eventually evolved into what is now

Curry College.) Since pure functional programming is centered around the notion of an *expression*, I thought that *The Haskell School of Expression* would be a good title for my first book. And it was thus quite natural to choose *The Haskell School of Music* for my second!

How To Read This Book

As mentioned earlier, there is a certain mind-set, a certain viewpoint of the world, and a certain approach to problem solving that collectively work best when programming in Haskell (this is true for any new programming paradigm). If you teach only Haskell language details to a C programmer, she is likely to write ugly, incomprehensible functional programs. But if you teach her how to think differently, how to see problems in a different light, functional solutions will come easily, and elegant Haskell programs will result. As Samuel Silas Curry once said:

> All expression comes *from within outward*, from the center to the surface, from a hidden source to outward manifestation. The study of expression as a natural process brings you into contact with cause and makes you feel the source of reality.

What is especially beautiful about this quote is that music is a kind of expression, although Curry was more likely talking about speech. In addition, as has been noted by many, music has many ties to mathematics. So for me, combining the elegant mathematical nature of Haskell with that of music is as natural as singing a nursery tune.

Using a high-level language to express musical ideas is, of course, not new. But Haskell is unique in its insistence on purity (no side effects), and this alone makes it particularly suitable for expressing musical ideas. By focusing on *what* a musical entity is rather than on *how* to create it, we allow musical ideas to take their natural form as Haskell expressions. Haskell's many abstraction mechanisms allow us to write musical programs that are elegant, concise, yet powerful. We will consistently attempt to let the music express itself as naturally as possible, without encoding it in terms of irrelevant language details.

Of course, my ultimate goal is to teach computer music concepts. But along the way you will also learn Haskell. There is no limit to what one might wish to do with compuiter music, and therefore the better you are at programming, the more success you will have. This is why I think that many languages designed specifically for computer music—although fun to work with, easy to use, and cute in concept—will ultimately be too limited in expressiveness.

My general approach to introducing computer music concepts is to first provide an intuitive explanation, then a mathematically rigorous definition, and finally fully executable Haskell code. In the process I will introduce Haskell features as they are needed, rather than all at once. I believe that this interleaving of concepts and applications makes the material easier to digest.

Another characteristic of my approach is that I won't hide any details—I want Haskore to be as transparent as possible! There are no magical built-in operations, no special computer music commands or values. This works out well for several reasons. First, there is in fact nothing ugly or difficult to hide—so why hide anything at all? Second, by reading the code, you will better and more quickly understand Haskell. Finally, by stepping through the design process with me, you may decide that you prefer a different approach—there is, after all, no One True Way to express computer music ideas. Indeed, I expect that this process will position you well to write rich, creative musical applications on your own.

I encourage the seasoned programmer having experience only with conventional imperative and/or object-oriented languages to read this text with an open mind. Many things will be different, and will likely feel awkward. There will be a tendency to rely on old habits when writing new programs, and to ignore suggestions about how to approach things differently. If you can manage to resist those tendencies I am confident that you will have an enjoyable learning experience. Many of those who succeed in this process find that many ideas about functional programming can be applied to imperative and object-oriented languages as well, and that their imperative coding style changes for the better.

I also ask the experienced programmer to be patient while in the earlier chapters I explain things like "syntax," "operator precedence," etc., since it is my goal that this text should be readable by someone having only modest prior programming experience. With patience the more advanced ideas will appear soon enough.

If you are a novice programmer, I suggest taking your time with the book; work through the exercises, and don't rush things. If, however, you don't fully grasp an idea, feel free to move on, but try to re-read difficult material at a later time when you have seen more examples of the concepts in action. For the most part this is a "show by example" textbook, and you should try to execute as many of the programs in this text as you can, as well as every program that you write. Learn-by-doing is the corollary to show-by-example.

Haskell Implementations

There are several good implementations of Haskell, all available free on the Internet through the Haskell Home Page at `http://haskell.org`. One that I especially recommend is *GHC*, an easy-to-use and easy-to-install Haskell compiler and interpreter (see `http://haskell.org/ghc`). GHC runs on a variety of platforms, including PC's (Windows XP and Vista), various flavors of Unix (Linux, FreeBSD, etc.), and Mac OS X. Any text editor can be used to create the source files, but I prefer to use emacs (see `http://www.gnu.org/software/emacs`), along with its Haskell mode (see `http://www.haskell.org/haskell-mode`). All of the source code from this textbook can be found at `http://plucky.cs.yale.edu/cs431`. Feel free to email me at `paul.hudak@yale.edu` with any comments, suggestions, or questions.

Happy Haskell Music Making!

Paul Hudak
New Haven
September 2008

Chapter 1

Computation by Calculation

Programming, in its broadest sense, is *problem solving.* It begins when we look out into the world and see problems that we want to solve, problems that we think can and should be solved using a digital computer. Understanding the problem well is the first—and probably the most important—step in programming, since without that understanding we may find ourselves wandering aimlessly down a dead-end alley, or worse, down a fruitless alley with no end. "Solving the wrong problem" is a phrase often heard in many contexts, and we certainly don't want to be victims of that crime. So the first step in programming is answering the question, "What problem am I trying to solve?"

Once you understand the problem, then you must find a solution. This may not be easy, of course, and in fact you may discover several solutions, so we also need a way to measure success. There are various dimensions in which to do this, including correctness ("Will I get the right answer?") and efficiency ("Will I have enough resources?"). But the distinction of which solution is better is not always clear, since the number of dimensions can be large, and programs will often excel in one dimension and do poorly in others. For example, there may be one solution that is fastest, one that uses the least amount of memory, and one that is easiest to understand. Deciding which to choose can be difficult, and is one of the more interesting challenges that you will face in programming.

The last measure of success mentioned above—clarity of a program—is somewhat elusive, most difficult to measure, and, quite frankly, sometimes difficult to rationalize. But in large software systems clarity is an especially important goal, since the most important maxim about such systems is that they are never really finished! The process of continuing work on a software system after it is delivered to users is what software engineers call *software maintenance*, and is the most expensive phase of the so-called "software lifecycle." Software maintenance includes fixing bugs in programs, as well as changing certain functionality and enhancing the system with new features in response to users' experience.

Therefore taking the time to write programs that are highly legible—easy to understand and reason about—will facilitate the software maintenance process. To complete the emphasis on this issue, it is important to realize that the person performing software maintenance is usually not the person who wrote the original program. So when you write your programs, write them as if you are writing them for someone else to see, understand, and ultimately pass judgement on!

As we work through the many musical examples in this book, we will sometimes express them in several different ways (some of which are dead-ends!), taking the time to contrast them in style, efficiency, clarity, and functionality.[1] We do this not just for pedagogical purposes. *Such reworking of programs is the norm*, and you are encouraged to get into the habit of doing so. Don't always be satisfied with your first solution to a problem, and always be prepared to go back and change—or even throw away—those parts of your program that you later discover do not fully satisfy your actual needs.

1.1 Computation by Calculation in Haskell

In this text we use the programming language *Haskell* to address many of the issues discussed in the last section. We will avoid the approach of explaining Haskell first and giving examples second. Rather, we will walk, step by step, along the path of understanding a problem, understanding the solution space, and understanding how to express a particular solution in Haskell. It is important to learn how to problem solve!

Along this path we will use whatever tools are appropriate for analyzing a particular problem, very often mathematical tools that should be familiar to the average college student, indeed most to the average high-school student. As we do this we will evolve our problems toward a particular view of computation that is especially useful: that of *computation by calculation.* You will find that such a viewpoint is not only powerful—we won't shy away from difficult problems—it is also *simple.* Haskell supports well the idea of computation by calculation. Programs in Haskell can be viewed as *functions*

[1] At times we will also explore different methods for proving properties of programs.

whose input is that of the problem being solved, and whose output is our desired result; and the behavior of functions can be understood easily as computation by calculation.

An example might help to demonstrate these ideas. Suppose we want to perform an arithmetic calculation such as $3 \times (9 + 5)$. In Haskell we would write this as $3 * (9 + 5)$, since most standard computer keyboards and text editors do not recognize the special symbol ×. To calculate the result, we proceed as follows:

$$\begin{array}{l} 3 * (9 + 5) \\ \Rightarrow 3 * 14 \\ \Rightarrow 42 \end{array}$$

It turns out that this is not the only way to compute the result, as evidenced by this alternative calculation:[2]

$$\begin{array}{l} 3 * (9 + 5) \\ \Rightarrow 3 * 9 + 3 * 5 \\ \Rightarrow 27 + 3 * 5 \\ \Rightarrow 27 + 15 \\ \Rightarrow 42 \end{array}$$

Even though this calculation takes two extra steps, it at least gives the correct answer. Indeed, an important property of each and every program in this textbook—in fact every program that can be written in the functional language Haskell—is that it will always yield the same answer when given the same inputs, regardless of the order we choose to perform the calculations.[3] This is precisely the mathematical definition of a function: for the same inputs, it always yields the same output.

On the other hand, the first calculation above took less steps than the second, and so we say that it is more *efficient*. Efficiency in both space (amount of memory used) and time (number of steps executed) is important when searching for solutions to problems, but of course if we get the wrong answer, efficiency is a moot point. In general we will search first for any solution to a problem, and later refine it for better performance.

The above calculations are fairly trivial, of course. But we will be doing much more sophisticated operations soon enough. For starters—and to

[2] This assumes that multiplication distributes over addition in the number system being used, a point that we will return to later.

[3] As long as we don't choose a non-terminating sequence of calculations, another issue that we will return to later.

introduce the idea of a function—we could *generalize* the arithmetic operations performed in the previous example by defining a function to perform them for any numbers x, y, and z:

$$\mathit{simple}\ x\ y\ z = x * (y + z)$$

This equation defines *simple* as a function of three *arguments*, x, y, and z. In mathematical notation, we might see the above written slightly differently, namely:

$$\mathit{simple}(x, y, z) = x \times (y + z)$$

In any case, it should be clear that "*simple* 3 9 5" is the same as "$3*(9+5)$." In fact the proper way to calculate the result is:

$$\begin{array}{l}\mathit{simple}\ 3\ 9\ 5\\ \Rightarrow 3 * (9 + 5)\\ \Rightarrow 3 * 14\\ \Rightarrow 42\end{array}$$

The first step in this calculation is an example of *unfolding* a function definition: 3 is substituted for x, 9 for y, and 5 for z on the right-hand side of the definition of *simple*. This is an entirely mechanical process, not unlike what the computer actually does to execute the program.

When we wish to say that an expression e evaluates (via zero, one, or possibly many more steps) to the value v, we will write $e \Longrightarrow v$ (this arrow is longer than that used earlier). So we can say directly, for example, that *simple* 3 9 5 $\Longrightarrow$ 42, which should be read "*simple* 3 9 5 evaluates to 42."

With *simple* now suitably defined, we can repeat the sequence of arithmetic calculations as often as we like, using different values for the arguments to *simple*. For example, *simple* 4 3 2 $\Longrightarrow$ 20.

We can also use calculation to *prove properties* about programs. For example, it should be clear that for any a, b, and c, *simple* $a\ b\ c$ should yield the same result as *simple* $a\ c\ b$. For a proof of this, we calculate *symbolically*; that is, using the symbols a, b, and c rather than concrete numbers such as 3, 5, and 9:

$$\begin{array}{l}\mathit{simple}\ a\ b\ c\\ \Rightarrow a * (b + c)\\ \Rightarrow a * (c + b)\\ \Rightarrow \mathit{simple}\ a\ c\ b\end{array}$$

We will use the same notation for these symbolic steps as for concrete ones. In particular, the arrow in the notation reflects the direction of our reasoning, and nothing more. In general, if $e1 \Rightarrow e2$, then it's also true that $e2 \Rightarrow e1$.

We will also refer to these symbolic steps as "calculations," even though the computer will not typically perform them when executing a program (although it might perform them *before* a program is run if it thinks that it might make the program run faster). The second step in the calculation above relies on the commutativity of addition (namely that, for any numbers x and y, $x + y = y + x$). The third step is the reverse of an unfold step, and is appropriately called a *fold* calculation. It would be particularly strange if a computer performed this step while executing a program, since it does not seem to be headed toward a final answer. But for proving properties about programs, such "backward reasoning" is quite important.

When we wish to make the justification for each step clearer, whether symbolic or concrete, we will present a calculation with more detail, as in:

$$
\begin{array}{l}
\mathit{simple}\ a\ b\ c \\
\Rightarrow \{\mathit{unfold}\} \\
a * (b + c) \\
\Rightarrow \{\mathit{commutativity}\} \\
a * (c + b) \\
\Rightarrow \{\mathit{fold}\} \\
\mathit{simple}\ a\ c\ b
\end{array}
$$

In most cases, however, this will not be necessary.

Proving properties of programs is another theme that will be repeated often in this text. As the world begins to rely more and more on computers to accomplish not just ordinary tasks such as writing term papers and sending email, but also life-critical tasks such as controlling medical procedures and guiding spacecraft, then the correctness of the programs that we write gains in importance. Proving complex properties of large, complex programs is not easy—and rarely if ever done in practice—but that should not deter us from proving simpler properties of the whole system, or complex properties of parts of the system, since such proofs may uncover errors, and if not, at least help us to gain confidence in our effort.

If you are someone who is already an experienced programmer, the idea of computing *everything* by calculation may seem odd at best, and naive at worst. How does one write to a file, draw a picture, play a sound, or respond to mouse-clicks? If you are wondering about these things, it is hoped that you have patience reading the early chapters, and that you find delight in reading the later chapters where the full power of this approach begins to shine. We will avoid, however, most comparisons between Haskell and conventional programming languages such as C, C++, Java, or even Scheme or ML (two "almost functional" languages).

In many ways this first chapter is the most difficult chapter in the entire text, since it contains the highest density of new concepts. If you have trouble with some of the ideas here, keep in mind that we will return to almost every idea at later points in the text. And don't hesitate to return to this chapter later to re-read difficult sections; they will likely be much easier to grasp at that time.

Exercise 1.1 Write out all of the steps in the calculation of the value of

$$\mathit{simple}\ (\mathit{simple}\ 2\ 3\ 4)\ 5\ 6$$

Exercise 1.2 Prove by calculation that $\mathit{simple}\ (a - b)\ a\ b \Longrightarrow a^2 - b^2$.

> **Details:** In the remainder of the text the need will often arise to explain some aspect of Haskell in more detail, without distracting too much from the primary line of discourse. In those circumstance we will off-set the comments and proceed them with the word "Details," such as is done with this paragraph.

1.2 Expressions, Values, and Types

In Haskell, the entities that we perform calculations on are called *expressions*, and the entities that result from a calculation—i.e. "the answers"—are called *values*. It is helpful to think of a value just as an expression on which no more calculation can be carried out.

Examples of expressions include *atomic* (meaning, indivisible) values such as the integer 42 and the character `'a'`, which are examples of two *primitive* atomic values. In the next Chapter we will will also see examples of *user-defined* atomic values, such as the pitch classes C, Cs, Df, etc. (denoting the musical notes C, C♯, D♭, etc.).

In addition, there are *structured* (meaning, made from smaller pieces) expressions such as the list $[\,C, Cs, Df\,]$ and the pair $(\texttt{'b'}, 4)$ (lists and pairs are different in a subtle way, to be described later). Each of these structured expressions is also a value, since by themselves there is no calculation that can be carried out. As another example, $1 + 2$ is an expression, and one step of calculation yields the expression 3, which is a value, since no more calculations can be performed.

Sometimes, however, an expression has only a never-ending sequence of calculations. For example, if x is defined as:

$$x = x + 1$$

then here's what happens when we try to calculate the value of x:

$$\begin{array}{l} x \\ \Rightarrow x + 1 \\ \Rightarrow (x + 1) + 1 \\ \Rightarrow ((x + 1) + 1) + 1 \\ \Rightarrow (((x + 1) + 1) + 1) + 1 \\ \ldots \end{array}$$

This is clearly a never-ending sequence of steps, in which case we say that the expression does not terminate, or is *non-terminating.* In such cases the symbol $\perp$, pronounced "bottom," is used to denote the value of the expression.

Every expression (and therefore every value) also has an associated *type.* You can think of types as sets of expressions (or values), in which members of the same set have much in common. Examples include the atomic types *Integer* (the set of all fixed-precision integers) and *Char* (the set of all characters), as well as the structured types [*Integer*] and [*PitchClass*] (the set of all lists of integers and pitch classes, respectively) and (*Char*, *Integer*) (the set of all character/integer pairs). The association of an expression or value with its type is very important, and there is a special way of expressing it in Haskell. Using the examples of values and types above, we write:

$$\begin{array}{r} 42 :: \mathit{Integer} \\ \texttt{'a'} :: \mathit{Char} \\ [\,C, \mathit{Cs}, \mathit{Df}\,] :: [\mathit{PitchClass}] \\ (\texttt{'b'}, 4) :: (\mathit{Char}, \mathit{Integer}) \end{array}$$

Details: Literal characters are written enclosed in single forward quotes, as in `'a'`, `'A'`, `'b'`, `','`, `'!'`, `' '` (a space), etc. (There are some exceptions, however; see the Haskell Report for details.)

The "::" should be read "has type," as in "42 has type *Integer*."

Details: Note that the names of specific types are capitalized, such as *Integer* and *Char*, but the names of values are not, such as *simple* and x. This is not just a convention: it is required when programming in Haskell.

> In addition, the case of the other characters matters, too. For example, test, teSt, and tEST are all distinct names for values, as are Test, TeST, and TEST for types.

Haskell's *type system* ensures that Haskell programs are *well-typed*; that is, that the programmer has not mismatched types in some way. For example, it does not make much sense to add together two characters, so the expression `'a'` + `'b'` is *ill-typed.* The best news is that Haskell's type system will tell you if your program is well-typed *before you run it.* This is a big advantage, since most programming errors are manifested as typing errors.

1.3 Function Types and Type Signatures

What should the type of a function be? It seems that it should at least convey the fact that a function takes values of one type—$\mathit{T1}$, say—as input, and returns values of (possibly) some other type—$\mathit{T2}$, say—as output. In Haskell this is written $\mathit{T1} \rightarrow \mathit{T2}$, and we say that such a function "maps values of type $\mathit{T1}$ to values of type $\mathit{T2}$." If there is more than one argument, the notation is extended with more arrows. For example, if our intent is that the function *simple* defined in the previous section has type $\mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}$, we can declare this fact by including a *type signature* with the definition of *simple*:

$$\begin{array}{l}\mathit{simple} :: \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \\ \mathit{simple}\ x\ y\ z = x * (y + z)\end{array}$$

> **Details:** When you write Haskell programs using a typical text editor, you will not see nice fonts and arrows as in $\mathit{Integer} \rightarrow \mathit{Integer}$. Rather, you will have to type `Integer -> Integer`.

Haskell's type system also ensures that user-supplied type signatures such as this one are correct. Actually, Haskell's type system is powerful enough to allow us to avoid writing any type signatures at all, in which case we say that the type system *infers* the correct types for us.[4] Nevertheless, judicious placement of type signatures, as we did for *simple*, is a good habit, since type signatures are an effective form of documentation and help bring programming errors to light. Also, in almost every example in this text, we

[4] There are a few exceptions this rule, and in the case of *simple* the inferred type is actually a bit more general than that written above. Both of these points will be returned to later.

will make a habit of first talking about the types of expressions and functions as a way to better understand the problem at hand, organize our thoughts, and lay down the first ideas of a solution.

The normal use of a function is referred to as *function application*. For example, *simple* 3 9 5 is the application of the function *simple* to the arguments 3, 9, and 5.

> **Details:** Some functions, such as $(+)$, are applied using what is known as *infix syntax*; that is, the function is written between the two arguments rather than in front of them (compare $x + y$ to $f\ x\ y$). Infix functions are often called *operators*, and are distinguished by the fact that they do not contain any numbers or letters of the alphabet. Thus $\hat{}!$ and $*\#:$ are infix operators, whereas *thisIsAFunction* and *f9g* are not (but are still valid names for functions or other values). The only exception to this is that the symbol $'$ is considered to be alphanumeric; thus f' and $one's$ are valid names, but not operators.
>
> In Haskell, when referring to an operator as a value, it is enclosed in parentheses, such as when declaring its type, as in:
>
> $$(+) :: \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}$$
>
> Also, when trying to understand an expression such as $f\ x + g\ y$, there is a simple rule to remember: function application always has "higher precedence" than operator application, so that $f\ x + g\ y$ is the same as $(f\ x) + (g\ y)$.
>
> Despite all of these syntactic differences, however, operators are still just functions.

Exercise 1.3 Identify the well-typed expressions in the following and, for each, give its proper type:

$[(2,3),(4,5)]$
$[\mathit{Cs}, 42]$
$(\mathit{Df}, -42)$
simple `'a'` `'b'` `'c'`
$(\mathit{simple}\ 1\ 2\ 3, \mathit{simple})$

1.4 Abstraction, Abstraction, Abstraction

The title of this section is the answer to the question: "What are the three most important ideas in programming?" Well, perhaps this is an overstatement, but the hope is that it has gotten your attention, at least. Webster defines the verb "abstract" as follows:

> **abstract**, *vt* (1) remove, separate (2) to consider apart from application to a particular instance.

In programming we do this when we see a repeating pattern of some sort, and wish to "separate" that pattern from the "particular instances" in which it appears. Let's refer to this process as the *abstraction principle*, and see how it might manifest itself in problem solving.

1.4.1 Naming

One of the most basic ideas in programming—for that matter, in every day life—is to *name* things. For example, we may wish to give a name to the value of π, since it is inconvenient to retype (or remember) the value of π beyond a small number of digits. In mathematics the greek letter π in fact *is* the name for this value, but unfortunately we don't have the luxury of using greek letters on standard computer keyboards and text editors. So in Haskell we write:

$$\begin{aligned}&pi :: \mathit{Float}\\&pi = 3.1415927\end{aligned}$$

to associate the name *pi* with the number 3.1415927. The type signature in the first line declares *pi* to be a *floating-point number*, which mathematically—and in Haskell—is distinct from an integer.[5] Now we can use the name *pi* in expressions whenever we want; it is an abstract representation, if you will, of the number 3.1415927. Furthermore, if we ever have a need to change a named value (which hopefully won't ever happen for *pi*, but could certainly happen for other values), we would only have to change it in one place, instead of in the possibly large number of places where it is used.

Suppose now that we are working on a problem whose solution requires writing some expression more than once. For example, we might find ourselves computing something such as:

$$\begin{aligned}&x :: \mathit{Float}\\&x = f\ (a - b + 2) + g\ y\ (a - b + 2)\end{aligned}$$

The first line declares x to be a floating-point number, while the second is an equation that defines the value of x. Note on the right-hand side of this equation that the expression $a - b + 2$ is repeated—it has two instances—and thus, applying the abstraction principle, we wish to separate it from

[5] We will have more to say about floating-point numbers later in this chapter.

these instances. We already know how to do this—it's called *naming*—so we might choose to rewrite the single equation above as two:

$$\begin{array}{l} c = a - b + 2 \\ x = f\ c + g\ y\ c \end{array}$$

If, however, the definition of c is not intended for use elsewhere in the program, then it is advantageous to "hide" the definition of c within the definition of x. This will avoid cluttering up the namespace, and prevents c from clashing with some other value named c. To achieve this, we simply use a **let** expression:

$$\begin{array}{l} x = \textbf{let}\ c = a - b + 2 \\ \qquad \textbf{in}\ f\ c + g\ y\ c \end{array}$$

A **let** expression restricts the *visibility* of the names that it creates to the internal workings of the **let** expression itself. For example, if we write:

$$\begin{array}{l} c = 42 \\ x = \textbf{let}\ c = a - b + 2 \\ \qquad \textbf{in}\ f\ c + g\ y\ c \end{array}$$

then there is no conflict of names—the "outer" c is completely different from the "inner" one enclosed in the **let** expression. Think of the inner c as analogous to the first name of someone in your household. If your brother's name is "John" he will not be confused with John Thompson who lives down the street when you say, "John spilled the milk."

Details: An equation such as $c = 42$ is called a *binding*. A simple rule to remember when programming in Haskell is never to give more than one binding for the same name in a context where the names can be confused, whether at the top level of your program or nestled within a **let** expression. For example, this is not allowed:

$$\begin{array}{l} a = 42 \\ a = 43 \end{array}$$

nor is this:

$$\begin{array}{l} a = 42 \\ b = 43 \\ a = 44 \end{array}$$

So you can see that naming—using either top-level equations or equations within a **let** expression—is an example of the abstraction principle in action.

1.4.2 Functional Abstraction

[I should replace the following with something musical, but for now it will have to do as is.]

Let's now consider a more complex example. Suppose we are computing the sum of the areas of three circles with radii *r1*, *r2*, and *r3*, as expressed by:

```
totalArea :: Float
totalArea = pi ∗ r1ˆ2 + pi ∗ r2ˆ2 + pi ∗ r3ˆ2
```

Details: (ˆ) is Haskell's integer exponentiation operator. In mathematics we would write $\pi \times r^2$ or just πr^2 instead of *pi* ∗ *r*ˆ2.

Although there isn't an obvious repeating expression here as there was in the last example, there is a repeating *pattern of operations*. Namely, the operations that square some given quantity—in this case the radius—and then multiply the result by π. To abstract a sequence of operations such as this, we use a *function*—which we will give the name *circleArea*—that takes the "given quantity"—the radius—as an argument. There are three instances of the pattern, each of which we can expect to replace with a call to *circleArea*. This leads to:

```
circleArea :: Float → Float
circleArea r = pi ∗ rˆ2

totalArea = circleArea r1 + circleArea r2 + circleArea r3
```

Using the idea of unfolding described earlier, it is easy to verify that this definition is equivalent to the previous one.

This application of the abstraction principle is sometimes called *functional abstraction*, since the sequence of operations is abstracted as a function, in this case *circleArea*. Actually, it can be seen as a generalization of the previous kind of abstraction: *naming*. That is, *circleArea r1* is just a name for *pi*∗*r1*ˆ2, *circleArea r2* for *pi*∗*r2*ˆ2, and *circleArea r3* for *pi*∗*r3*ˆ2. Or in other words, a named quantity such as *c* or *pi* defined previously can be thought of as a function with no arguments.

Note that *circleArea* takes a radius (a floating-point number) as an argument and returns the area (also a floating-point number) as a result, as reflected in its type signature.

The definition of *circleArea* could also be hidden within *totalArea* using a **let** expression as we did in the previous example:

```
totalArea = let circleArea r = pi * r^2
            in circleArea r1 + circleArea r2 + circleArea r3
```

On the other hand, it is more likely that computing the area of a circle will be useful elsewhere in the program, so leaving the definition at the top level is probably preferable in this case.

1.4.3 Data Abstraction

The value of *totalArea* is the sum of the areas of three circles. But what if in another situation we must add the areas of five circles, or in other situations even more? In situations where the number of things is not certain, it is useful to represent them in a *list* whose length is arbitrary. So imagine that we are given an entire list of circle areas, whose length isn't known at the time we write the program. What now?

We will define a function *listSum* to add the elements of a list. Before doing so, however, there is a bit more to say about lists.

Lists are an example of a *data structure*, and when their use is motivated by the abstraction principle, we will say that we are applying *data abstraction*. Earlier we saw the example $[1, 2, 3]$ as a list of integers, whose type is thus $[\mathit{Integer}]$. A list with *no* elements is—not surprisingly—written *[]*, and pronounced "nil." To add a single element x to the front of a list xs, we write $x : xs$. (Note the naming convention used here; xs is the plural of x, and should be read that way.) In fact, the list $[1, 2, 3]$ is equivalent to $1 : (2 : (3 : [\,]))$, which can also be written $1 : 2 : 3 : [\,]$ since the infix operator *(:)* is "right associative."

Details: In mathematics we rarely worry about whether the notation $a + b + c$ stands for $(a + b) + c$ (in which case $+$ would be "left associative") or $a + (b + c)$ (in which case $+$ would "right associative"). This is because in situations where the parentheses are left out it's usually the case that the operator is *mathematically* associative, meaning that it doesn't matter which interpretation we choose. If the interpretation *does* matter, mathemeticians will include parentheses to make it clear. Furthermore, in mathematics there is an implicit assumption that some operators have higher *precedence* than others; for example, $2 \times a + b$ is interpreted as $(2 \times a) + b$, not $2 \times (a + b)$.

In most programming languages, including Haskell, each operator is defined as having some precedence level and to be either left or right associative. For arithmetic operators, mathematical convention is usually followed; for example, $2 * a + b$ is interpreted as $(2 * a) + b$ in Haskell. The predefined list-forming operator (:) is defined to be right associative. Just

> as in mathematics, this associativey can be over-ridden by using parentheses: thus $(a : b) : c$ is a valid Haskell expression (assuming that it is well-typed), and is very different from $a : b : c$. We will see later how to specify the associativity and precedence of new operators that we define.

Examples of pre-defined functions defined on lists in Haskell include *head* and *tail*, which return the "head" and "tail" of a list, respectively. That is, $head\ (x : xs) \Rightarrow x$ and $tail\ (x : xs) \Rightarrow xs$ (we will define these two functions formally in Section 3.1). Another example is the function $(+\!\!+)$ which *concatenates*, or *appends*, together its two list arguments. For example, $[1, 2, 3] +\!\!+ [4, 5, 6] \Rightarrow [1, 2, 3, 4, 5, 6]$ ($(+\!\!+)$ will be defined in Section 3.3).

Returning to the problem of defining a function to add the elements of a list, let's first express what its type should be:

$$listSum :: [Float] \rightarrow Float$$

Now we must define its behavior appropriately. Often in solving problems such as this it is helpful to consider, one by one, all possible cases that could arise. To compute the sum of the elements of a list, what might the list look like? The list could be empty, in which case the sum is surely 0. So we write:

$$listSum\ [\,] = 0$$

The other possibility is that the list *isn't* empty—i.e. it contains at least one element—in which case the sum is the first number plus the sum of the remainder of the list. So we write:

$$listSum\ (x : xs) = x + listSum\ xs$$

Combining these two equations with the type signature brings us to the complete definition of the function *listSum*:

$$\begin{array}{l} listSum :: [Float] \rightarrow Float \\ listSum\ [\,] = 0 \\ listSum\ (x : xs) = x + listSum\ xs \end{array}$$

> **Details:** Although intuitive, this example highlights an important aspect of Haskell: *pattern matching*. The left-hand sides of the equations contain *patterns* such as $[\,]$ and $x : xs$. When a function is applied, these patterns are *matched* against the argument values in a fairly intuitive way ($[\,]$ only matches the empty list, and $x : xs$ will successfully match any list with at

> least one element, while naming the first element x and the rest of the list xs). If the match succeeds, the right-hand side is evaluated and returned as the result of the application. If it fails, the next equation is tried, and if all equations fail, an error results. All of the equations that define a particular function must appear together, one after the other.
>
> Defining functions by pattern matching is quite common in Haskell, and you should eventually become familiar with the various kinds of patterns that are allowed; see Appendix D for a concise summary.

This is called a *recursive* function definition since *listSum* "refers to itself" on the right-hand side of the second equation. Recursion is a very powerful technique that you will see used many times in this text. It is also an example of a general problem-solving technique where a large problem is broken down into many simpler but similar problems; solving these simpler problems one-by-one leads to a solution to the larger problem.

Here is an example of *listSum* in action:

$$\begin{array}{l}
\mathit{listSum}\ [1,2,3] \\
\Rightarrow \mathit{listSum}\ (1 : (2 : (3 : [\,]))) \\
\Rightarrow 1 + \mathit{listSum}\ (2 : (3 : [\,])) \\
\Rightarrow 1 + (2 + \mathit{listSum}\ (3 : [\,])) \\
\Rightarrow 1 + (2 + (3 + \mathit{listSum}\ [\,])) \\
\Rightarrow 1 + (2 + (3 + 0)) \\
\Rightarrow 1 + (2 + 3) \\
\Rightarrow 1 + 5 \\
\Rightarrow 6
\end{array}$$

The first step above is not really a calculation, but rather a rewriting of the list syntax. The remaining calculations consist of four unfold steps followed by three integer additions.

Given this definition of *listSum* we can rewrite the definition of *totalArea* as:

$$\mathit{totalArea} = \mathit{listSum}\ [\mathit{circleArea}\ \mathit{r1}, \mathit{circleArea}\ \mathit{r2}, \mathit{circleArea}\ \mathit{r3}]$$

This may not seem like much of an improvement, but if we were adding many such circle areas in some other context, it would be. Indeed, lists are arguably the most commonly used structured data type in Haskell. In the next chapter we will see a more convincing example of the use of lists; namely, to represent the vertices that make up a polygon. Since a polygon can have an arbitrary number of vertices, using a data structure such as a list seems like just the right approach.

In any case, how do we know that this version of *totalArea* behaves the same as the original one? By calculation, of course:

$$
\begin{array}{l}
\mathit{listSum}\ [\,\mathit{circleArea}\ \mathit{r1}, \mathit{circleArea}\ \mathit{r2}, \mathit{circleArea}\ \mathit{r3}\,]\\
\Longrightarrow \{\,\mathit{unfold\ listSum}\ (\mathit{four\ succesive\ times})\,\}\\
\mathit{circleArea}\ \mathit{r1} + \mathit{circleArea}\ \mathit{r2} + \mathit{circleArea}\ \mathit{r3} + 0\\
\Longrightarrow \{\,\mathit{unfold\ circleArea}\ (\mathit{three\ places})\,\}\\
\mathit{pi} * \mathit{r1}\,\hat{}\,2 + \mathit{pi} * \mathit{r2}\,\hat{}\,2 + \mathit{pi} * \mathit{r3}\,\hat{}\,2 + 0\\
\Rightarrow \{\,\mathit{simple\ arithmetic}\,\}\\
\mathit{pi} * \mathit{r1}\,\hat{}\,2 + \mathit{pi} * \mathit{r2}\,\hat{}\,2 + \mathit{pi} * \mathit{r3}\,\hat{}\,2
\end{array}
$$

1.5 Code Reuse and Modularity

There doesn't seem to be much repetition in our last definition for *totalArea*, so perhaps we're done. In fact, let's pause for a moment and consider how much progress we've made. We started with the definition:

$$\mathit{totalArea} = \mathit{pi} * \mathit{r1}\,\hat{}\,2 + \mathit{pi} * \mathit{r2}\,\hat{}\,2 + \mathit{pi} * \mathit{r3}\,\hat{}\,2$$

and ended with:

$$\mathit{totalArea} = \mathit{listSum}\ [\,\mathit{circleArea}\ \mathit{r1}, \mathit{circleArea}\ \mathit{r2}, \mathit{circleArea}\ \mathit{r3}\,]$$

But additionally, we have introduced definitions for the auxiliary functions *circleArea* and *listSum*. In terms of size, our final program is actually larger than what we began with! So have we actually improved things?

From the standpoint of "removing repeating patterns," we certainly have, and we could argue that the resulting program is easier to understand as a result. But there is more. Now that we have defined auxiliary functions such as *circleArea* and *listSum*, we can *reuse* them in other contexts. Being able to reuse code is also called *modularity*, since the reused components are like little modules, or bricks, that can form the foundation of many applications.[6] We've already talked about reusing *circleArea*; and *listSum* is surely reusable: imagine a list of grocery item prices, or class sizes, or city populations, for each of which we must compute the total. In later chapters you will learn other concepts—most notably higher-order functions and polymorphism—that will substantially increase your ability to reuse code.

[6] "Code reuse" and "modularity" are important software engineering principles.

1.6 Beware of Programming with Numbers

In mathematics there are many different kinds of number systems. For example, there are integers, natural numbers (i.e. non-negative integers), real numbers, rational numbers, and complex numbers. These number systems possess many useful properties, such as the fact that multiplication and addition are commutative, and that multiplication distributes over addition. You have undoubtedly learned many of these properties in your studies, and have used them often in algebra, geometry, trigonometry, physics, etc.

Unfortunately, each of these number systems places great demands on computer systems. In particular, a number can in general require an *arbitrary amount of memory* to represent it—even an infinite amount! Clearly, for example, we cannot represent an irrational number such as π exactly; the best we can do is approximate it, or possibly write a program that computes it to whatever (finite) precision that we need in a given application. But even integers (and therefore rational numbers) present problems, since any given integer can be arbitrarily large.

Most programming languages do not deal with these problems very well. In fact, most programming languages do not have exact forms of any of these number systems. Haskell does slightly better than most, in that it has exact forms of integers (the type *Integer*) as well as rational numbers (the type *Rational*, defined in the Ratio Library). But in Haskell and most other languages there is no exact form of real numbers, for example, which are instead approximated by *floating-point numbers* with either single-word precision (*Float* in Haskell) or double-word precision (*Double*). What's worse, the behavior of arithmetic operations on floating-point numbers can vary somewhat depending on what CPU is being used, although hardware standardization in recent years has reduced the degree of this problem.

The bottom line is that, as simple as they may seem, great care must be taken when programming with numbers. Many computer errors, some quite serious and renowned, were rooted in numerical incongruities. The field of mathematics known as *numerical analysis* is concerned precisely with these problems, and programming with floating-point numbers in sophisticated applications often requires a good understanding of numerical analysis to devise proper algorithms and write correct programs.

As a simple example of this problem, consider the distributive law, expressed here as a calculation in Haskell and used earlier in this chapter in calculations involving the function *simple*:

$$a * (b + c) \Rightarrow a * b + a * c$$

For most floating-point numbers, this law is perfectly valid. For example, in the GHC implementation of Haskell, the expressions $pi * (3 + 4) :: \mathit{Float}$ and $pi * 3 + pi * 4 :: \mathit{Float}$ both yield the same result: 21.99115. But funny things can happen when the magnitude of $b + c$ differs significantly from the magnitude of either b or c. For example, the following two calculations are from GHC:

$$5 * (-0.123456 + 0.123457) :: \mathit{Float} \Rightarrow 4.991889\mathrm{e} - 6$$
$$5 * (-0.123456) + 5 * (0.123457) :: \mathit{Float} \Rightarrow 5.00679\mathrm{e} - 6$$

Although the error here is small, its very existence is worrisome, and in certain situations it could be disastrous. We will not discuss the nature of floating-point numbers much further in this text, but just remember that they are *approximations* to the real numbers. If real-number accuracy is important to your application, further study of the nature of floating-point numbers is probably warranted.

On the other hand, the distributive law (and many others) is valid in Haskell for the exact data types *Integer* and *Ratio Integer* (i.e. rationals). However, another problem arises: although the representation of an *Integer* in Haskell is not normally something that we are concerned about, it should be clear that the representation must be allowed to grow to an arbitrary size. For example, Haskell has no problem with the following number:

$$\begin{array}{l} \mathit{veryBigNumber} :: \mathit{Integer} \\ \mathit{veryBigNumber} = 43208345720348593219876512372134059 \end{array}$$

and such numbers can be added, multiplied, etc. without any loss of accuracy. However, such numbers cannot fit into a single word of computer memory, most of which are limited to 32 bits. Worse, since the computer system does not know ahead of time exactly how many words will be required, it must devise a dynamic scheme to allow just the right number of words to be used in each case. The overhead of implementing this idea unfortunately causes programs to run slower.

For this reason, Haskell provides another integer data type called *Int* which has maximum and minimum values that depend on the word-size of the CPU being used. In other words, every value of type *Int* fits into one word of memory, and the primitive machine instructions for integers can be used to manipulate them very efficiently.[7] Unfortunately, this means

[7] The Haskell Report requires that every implementation support *Int*s in the range -2^{29} to $2^{29} - 1$, inclusive. The GHC implementation running on a Pentium processor, for example, supports the range -2^{31} to $2^{31} - 1$.

that *overflow* or *underflow* errors could occur when an *Int* value exceeds either the maximum or minumum values. However, most implementations of Haskell (as well as most other languages) do not even tell you when this happens. For example, in GHC, the following *Int* value:

$$i :: Int$$
$$i = 1234567890$$

works just fine, but if you multiply it by two, GHC returns the value −1825831516! This is because twice *i* exceeds the maximum allowed value, so the resulting bits become nonsensical,[8] and are interpreted in this case as a negative number of the given magnitude.

This is alarming! Indeed, why should anyone ever use *Int* when *Integer* is available? The answer, as mentioned earlier, is efficiency, but clearly care should be taken when making this choice. If you are indexing into a list, for example, and you are confident that you are not performing index calculations that might result in the above kind of error, then *Int* should work just fine, since a list longer than 2^{31} will not fit into memory anyway! But if you are calculating the number of microseconds in some large time interval, or counting the number of people living on earth, then *Integer* would most likely be a better choice. Choose your number data types wisely!

In this text we will use the data types *Integer*, *Int*, *Float*, *Double* and *Rational* for a variety of different applications; for a discussion of the other number types, consult the Haskell Report. As we use these data types, we will do so without much discussion—this is not, after all, a book on numerical analysis—but we will issue a warning whenever reasoning about floating-point numbers, for example, in a way that might not be technically sound.

[8] Actually, they are perfectly sensible in the following way: the 32-bit binary representation of *i* is 01001001100101100000001011010010, and twice that is 10010011001011000000010110100100. But the latter number is seen as negative because the 32nd bit (the highest-order bit on the CPU on which this was run) is a one, which means it is a negative number in "twos-complement" representation. The twos-complement of this number is in turn 01101100110100111111101001011100, whose decimal representation is 1825831516.

Chapter 2

Simple Music

```
module Haskore.Music where
infixr 5:+:, :=:
```

In the previous chapter we introduced some of the fundamental ideas of functional programming in Haskell. In this chapter we begin to develop some *musical* ideas as well. As we do so, more Haskell features will be introduced.

2.1 Preliminaries

Sometimes it is useful to use a built-in Haskell data type to directly represent some concept of interest. For example, we may wish to use *Int* to represent octaves, where by convention octave 4 corresponds to the octave containing middle C on the piano. We can express this in Haskell using a *type synonym*:

```
type Octave = Int
```

A type synonym does not create a new data type—it just gives a new name to an existing type. Type synonyms can be defined not just for atomic types such as *Int*, but also for structured types such as pairs. For example, in music theory a pitch is normally defined as a pair, a pitch class and an octave. Assuming the existence of a data type called *PitchClass*, we can write the following type synonym:

```
type Pitch = (PitchClass, Octave)
```

For example, "concert A," i.e. A above middle C (sometimes written A4) corresponds to the pitch $(A, 4)$. For convenience we could define a Haskell variable with that value as follows:

```
a4 :: Pitch
a4 = (A, 4)      -- concert A
```

> **Details:** This example also demonstrates the use of program *comments*. Any text to the right of “ -- ” till the end of the line is considered to be a comment, and is effectively ignored. Haskell also permits *nested* comments that have the form {- this is a comment -} and can appear anywhere in a program.

Another useful musical concept is *duration*. Rather than use either integers or floating-point numbers, we will use *rational* numbers to denote duration:

```
type Dur = Rational
```

Rational is the data type of rational numbers expressed as ratios of *Integer*s in Haskell.

Rational numbers in Haskell are printed by GHCi in the form $n \% d$, where n is the numerator, and d is the denominator. Even a whole number, say the number 42, will print as $42 \% 1$ if it is a *Rational* number. To create a *Rational* number in our program, however, all we have to do is use the normal division operator, as in the following definition of a a quarter note:

```
qn :: Dur
qn = 1 / 4      -- quarter note
```

So far so good. But what about *PitchClass*? We might try to use integers to represent pitch classes as well, but this is not very elegant—ideally we would like to write something that looks more like the conventional pitch class names C, C♯, D♭, D, etc. The solution is to use an *algebraic data type* in Haskell:

```
data PitchClass = Cff | Cf | C | Dff | Cs | Df | Css | D | Eff | Ds
                | Ef | Fff | Dss | E | Es | Ff | F | Gff | Ess | Fs
                | Gf | Fss | G | Aff | Gs | Af | Gss | A | Bff | As
                | Bf | Ass | B | Bs | Bss
     deriving (Eq, Ord, Show, Read, Enum)
```

Ignoring the line beginning with “**deriving**” for the moment, this data type declaration simply enumerates the 21 pitch class names (three for each of the note names A through G). Note that enharmonics (such as G♯ and A♭) are listed separately, which may be important in certain applications.

> **Details:** All constructors in a **data** declaration must be capitalized. In this way they are syntactically distinguished from ordinary values. This distinction is useful since only constructors can be used in the pattern matching that is part of a function definition, as will be described shortly.

Keep in mind that *PitchClass* is a completely new, user-defined data type that is not equal to any other.

2.2 Notes and Music

We can of course define other data types for other purposes. For example, we will want to define the notion of a *note* (the pairing of a pitch with a duration), and a *rest*. Both of these can be thought of as *primitive* musical values, and thus we write:

```
data Prim = Note Dur Pitch
          | Rest Dur
     deriving (Show, Eq, Ord)
```

For example, *Note qn a4* is concert A played as a quarter note, and *Rest* 1 is a whole-note rest.

This definition is not completely satisfactory, however, because we may wish to attach other information to a note, such as its loudness, or some other annotation or articulation. Furthermore, the pitch itself may actually be a percussive sound, having no true pitch at all. To fix this we will introduce an important concept in Haskell, namely *polymorphism*—the ability to parameterize over types. Instead of fixing the type of the pitch of a note, we will leave it unspecified through the use of a *type variable*, as follows:

```
data Primitive a = Note Dur a
                 | Rest Dur
     deriving (Show, Eq, Ord)
```

Note the type variable *a*, which is used as an argument to *Primitive*, and then used in the body of the declaration—just like a variable in a function. *Primitive Pitch* is now the same as (or, technically, is now *isomorphic to*) the type *Prim*. Indeed, instead of defining *Prim* as above, we could now use a type synonym instead:

```
type Prim = Primitive Pitch
```

But *Primitive* is more flexible than *Prim*, since, for example, we could add loudness by pairing loudness with pitch, as in *Primitive* (*Pitch*, *Loudness*). We will see more concrete instances of this idea later.

So far we only have a way to express primitive notes and rests—how do we combine many notes and rests into a larger composition? To achieve this we will define another polymorphic data type, perhaps the most important data type used in this book, which defines the fundamental structure of a musical entity:

```
data Music a = Primitive (Primitive a)      -- primitive value
             | Music a :+: Music a          -- sequential composition
             | Music a :=: Music a          -- parallel composition
             | Modify Control (Music a)     -- modifier
    deriving (Show, Eq, Ord)
```

Details: The first line here looks odd: the name *Primitive* appears twice. The first occurence, however, is the name of a new *constructor* in the *Music* data type, whereas the second is the name of the existing *data type* defined above. Haskell allows using the same name to define a constructor and a data type, since they can never be confused: the context in which they are used will always be sufficient to distinguish them.

Also note the use of *infix constructors* (:+:) and (:=:). Infix constructors are just like infix operators in Haskell, but they must begin with a colon. This distinction exists to make it easier to pattern match, and is analogous to the distinction between ordinary names (which must begin with a lowercase character) and constructor names (which must begin with an uppercase character).

It is convenient to represent these musical ideas as a recursive datatype because we wish to not only construct musical values, but also take them apart, analyze their structure, print them in a structure-preserving way, interpret them for performance purposes, etc. We will see many examples of these kinds of processes shortly.

This data type declaration essentially says that a value of type *Music a* has one of four possible forms:

- *Primitive p*, where p is a primitive value of type *Primitive a*, for some type *a*. For example:

  ```
  ma4 :: Music Pitch
  ma4 = Primitive (Note qn a4)
  ```

is the musical value corresponding to a quarter-note rendition of concert A.

- *m1* :+: *m2* is the *sequential* composition of *m1* and *m2*; i.e. *m1* and *m2* are played in sequence.
- *m1* :=: *m2* is the *parallel* composition of *m1* and *m2*; i.e. *m1* and *m2* are played simultaneously.
- *Modify cntrl m* is an "annotated" version of *m* in which the control parameter *cntrl* specifies some way in which *m* is to be modified.

> **Details:** Note that *Music a* is defined in terms of *Music a*, and thus we say that is a *recursive* data type. It is also often called an *inductive* data type, since it is, in essence, an inductive definition of an infinite number of values, each of which can be arbitrarily complex.

The *Control* data type is defined as follows:

```
data Control =
        Tempo Rational             -- scale the tempo
      | Transpose AbsPitch         -- transposition
      | Instrument InstrumentName     -- intrument label
      | Phrase [PhraseAttribute]      -- phrase attributes
      | Player PlayerName          -- player label
    deriving (Show, Eq, Ord)

type PlayerName = String
```

It allows one to annotate a *Music* value with a tempo change, a transposition, a phrase attribute, a player name, or an instrument. Instrument names are borrowed from the General MIDI standard, and are captured as an algebraic data type in Figure 2.1. Phrase attributes and the concept of a "player" are closely related, but a full explanation is deferred until Chapter 6.

2.3 Convenient Auxiliary Functions

For convenient we define a number of functions to make it easier to write certain kinds of musical values. For starters, we define:

```
note d p = Primitive (Note d p)
rest d = Primitive (Rest d)
```

```
data InstrumentName
  = AcousticGrandPiano | BrightAcousticPiano | ElectricGrandPiano
  | HonkyTonkPiano | RhodesPiano | ChorusedPiano
  | Harpsichord | Clavinet | Celesta | Glockenspiel | MusicBox
  | Vibraphone | Marimba | Xylophone | TubularBells
  | Dulcimer | HammondOrgan | PercussiveOrgan
  | RockOrgan | ChurchOrgan | ReedOrgan
  | Accordion | Harmonica | TangoAccordion
  | AcousticGuitarNylon | AcousticGuitarSteel | ElectricGuitarJazz
  | ElectricGuitarClean | ElectricGuitarMuted | OverdrivenGuitar
  | DistortionGuitar | GuitarHarmonics | AcousticBass
  | ElectricBassFingered | ElectricBassPicked | FretlessBass
  | SlapBass1 | SlapBass2 | SynthBass1 | SynthBass2
  | Violin | Viola | Cello | Contrabass | TremoloStrings
  | PizzicatoStrings | OrchestralHarp | Timpani
  | StringEnsemble1 | StringEnsemble2 | SynthStrings1
  | SynthStrings2 | ChoirAahs | VoiceOohs | SynthVoice
  | OrchestraHit | Trumpet | Trombone | Tuba
  | MutedTrumpet | FrenchHorn | BrassSection | SynthBrass1
  | SynthBrass2 | SopranoSax | AltoSax | TenorSax
  | BaritoneSax | Oboe | Bassoon | EnglishHorn | Clarinet
  | Piccolo | Flute | Recorder | PanFlute | BlownBottle
  | Shakuhachi | Whistle | Ocarina | Lead1Square
  | Lead2Sawtooth | Lead3Calliope | Lead4Chiff
  | Lead5Charang | Lead6Voice | Lead7Fifths
  | Lead8BassLead | Pad1NewAge | Pad2Warm
  | Pad3Polysynth | Pad4Choir | Pad5Bowed
  | Pad6Metallic | Pad7Halo | Pad8Sweep
  | FX1Train | FX2Soundtrack | FX3Crystal
  | FX4Atmosphere | FX5Brightness | FX6Goblins
  | FX7Echoes | FX8SciFi | Sitar | Banjo | Shamisen
  | Koto | Kalimba | Bagpipe | Fiddle | Shanai
  | TinkleBell | Agogo | SteelDrums | Woodblock | TaikoDrum
  | MelodicDrum | SynthDrum | ReverseCymbal
  | GuitarFretNoise | BreathNoise | Seashore
  | BirdTweet | TelephoneRing | Helicopter
  | Applause | Gunshot | Percussion
  | Custom String
  deriving (Show, Eq, Ord)
```

Figure 2.1: General MIDI Instrument Names

```
tempo r m = Modify (Tempo r) m
transpose i m = Modify (Transpose i) m
instrument i m = Modify (Instrument i) m
phrase pa m = Modify (Phrase pa) m
player pn m = Modify (Player pn) m
```

We can also create simple names for familiar notes, durations, and rests, as shown in Figures 2.2 and 2.3. Despite the large number of them, these names are sufficiently “unusual” that name clashes are unlikely.

As a simple example, here is a ii-V-I chord progression in C major:

```
t251 :: Music Pitch
t251 = let dMinor = d 3 wn :=: f 3 1 :=: a 3 wn
           gMajor = g 3 wn :=: b 3 1 :=: d 4 wn
           cMajor = c 3 bn :=: e 3 2 :=: g 3 bn
       in dMinor :+: gMajor :+: cMajor
```

Details: Note that more than one equation is allowed in a **let** expression. The first characters of each equation, however, must line up vertically, and if an equation takes more than one line then the subsequent lines must be to the right of the first characters. For example, this is legal:

```
let a = aLongName
        + anEvenLongerName
    b = 56
in...
```

but neither of these are:

```
let a = aLongName
   +anEvenLongerName
     b = 56
in...

let a = aLongName
        + anEvenLongerName
   b = 56
in...
```

(The second line of the first example is too far to the left, as is the third line in the second example.)

Although this rule, called the *layout rule*, may seem a bit *ad hoc*, it avoids having to use special syntax to denote the end of one equation and the

```
cf, c, cs, df, d, ds, ef, e, es, ff, f, fs, gf, g, gs, af, a, as, bf, b, bs::
    Octave → Dur → Music Pitch

cff o d = note d (Cff, o)
cf o d = note d (Cf, o)
c o d = note d (C, o)
cs o d = note d (Cs, o)
css o d = note d (Css, o)
dff o d = note d (Dff, o)
df o d = note d (Df, o)
d o d = note d (D, o)
ds o d = note d (Ds, o)
dss o d = note d (Dss, o)
eff o d = note d (Eff, o)
ef o d = note d (Ef, o)
e o d = note d (E, o)
es o d = note d (Es, o)
ess o d = note d (Ess, o)
fff o d = note d (Fff, o)
ff o d = note d (Ff, o)
f o d = note d (F, o)
fs o d = note d (Fs, o)
fss o d = note d (Fss, o)
gff o d = note d (Gff, o)
gf o d = note d (Gf, o)
g o d = note d (G, o)
gs o d = note d (Gs, o)
gss o d = note d (Gss, o)
aff o d = note d (Aff, o)
af o d = note d (Af, o)
a o d = note d (A, o)
as o d = note d (As, o)
ass o d = note d (Ass, o)
bff o d = note d (Bff, o)
bf o d = note d (Bf, o)
b o d = note d (B, o)
bs o d = note d (Bs, o)
bss o d = note d (Bss, o)
```

Figure 2.2: Convenient note names.

```
bn, wn, hn, qn, en, sn, tn, sfn, dwn, dhn,
   dqn, den, dsn, dtn, ddhn, ddqn, dden :: Dur

bnr, wnr, hnr, qnr, enr, snr, tnr, dwnr, dhnr,
    dqnr, denr, dsnr, dtnr, ddhnr, ddqnr, ddenr :: Music Pitch

bn = 2; bnr = rest bn       -- brevis rest
wn = 1; wnr = rest wn       -- whole note rest
hn = 1 / 2; hnr = rest hn       -- half note rest
qn = 1 / 4; qnr = rest qn       -- quarter note rest
en = 1 / 8; enr = rest en       -- eight note rest
sn = 1 / 16; snr = rest sn       -- sixteenth note rest
tn = 1 / 32; tnr = rest tn       -- thirty-second note rest
sfn = 1 / 64; sfnr = rest sfn       -- sixty-fourth note rest

dwn = 3 / 2; dwnr = rest dwn       -- dotted whole note rest
dhn = 3 / 4; dhnr = rest dhn       -- dotted half note rest
dqn = 3 / 8; dqnr = rest dqn       -- dotted quarter note rest
den = 3 / 16; denr = rest den       -- dotted eighth note rest
dsn = 3 / 32; dsnr = rest dsn       -- dotted sixteenth note rest
dtn = 3 / 64; dtnr = rest dtn       -- dotted thirty-second note rest

ddhn = 7 / 8; ddhnr = rest ddhn       -- double-dotted half note rest
ddqn = 7 / 16; ddqnr = rest ddqn       -- double-dotted quarter note rest
dden = 7 / 32; ddenr = rest dden       -- double-dotted eighth note rest
```

Figure 2.3: Convenient rest names.

> beginning of the next (such as a semicolon), thus enhancing readability. In practice, use of layout is rather intuitive. Just remember two things:
>
> First, the first character following either **where** or **let** (and a few other keywords that we will see later) is what determines the starting column for the set of equations being written. Thus we can begin the equations on the same line as the keyword, the next line, or whatever.
>
> Second, just be sure that the starting column is further to the right than the starting column associated with any immediately surrounding clause (otherwise it would be ambiguous). The "termination" of an equation happens when something appears at or to the left of the starting column associated with that equation.

In order to play this simple example, we can import the *play* function from Hasore's MIDI library, and simply type:

```
play t251
```

at the GHC command line. Default instruments and tempos are used to then play the resulting composition.

2.4 Absolute Pitches

Treating pitches simply as integers is useful in many settings, so let's use a type synonym to introduce a concept of "absolute pitch:"

```
type AbsPitch = Int
```

The absolute pitch of a (relative) pitch can be defined mathematically as 12 times the octave, plus the index of the pitch class. We can express this in Haskell as follows:

```
absPitch :: Pitch → AbsPitch
absPitch (pc, oct) = 12 ∗ oct + pcToInt pc
```

> **Details:** Note the use of *pattern-matching* to match the argument of *absPitch* to a pair.

pcToInt is simply a function that converts a particular pitch class to an index, easily expressed as:

```
pcToInt :: PitchClass → Int
pcToInt Cff = −2
pcToInt Cf = −1
pcToInt C = 0
pcToInt Cs = 1
pcToInt Css = 2

pcToInt Dff = 0
pcToInt Df = 1
pcToInt D = 2
pcToInt Ds = 3
pcToInt Dss = 4

pcToInt Eff = 2
pcToInt Ef = 3
pcToInt E = 4
pcToInt Es = 5
pcToInt Ess = 6

pcToInt Fff = 3
pcToInt Ff = 4
pcToInt F = 5
pcToInt Fs = 6
pcToInt Fss = 7

pcToInt Gff = 5
pcToInt Gf = 6
pcToInt G = 7
pcToInt Gs = 8
pcToInt Gss = 9

pcToInt Aff = 7
pcToInt Af = 8
pcToInt A = 9
pcToInt As = 10
pcToInt Ass = 11

pcToInt Bff = 9
pcToInt Bf = 10
pcToInt B = 11
pcToInt Bs = 12
pcToInt Bss = 13
```

Converting an absolute pitch to a pitch is a bit more tricky, because of enharmonic equivalences. For example, the absolute pitch 15 might correspond to either $(Ds, 1)$ or $(Ef, 1)$. We take the approach of always returning a sharp in such ambiguous cases:

```
pitch :: AbsPitch → Pitch
pitch ap =
    let (oct, n) = divMod ap 12
    in ([C, Cs, D, Ds, E, F, Fs, G, Gs, A, As, B] !! n, oct)
```

> **Details:** (!!) is Haskell's zero-based list-indexing function; *list* !! n returns the $(n + 1)$th element in *list*. *divMod* x n returns a pair (q, r), where q is the integer quotient of x divided by n, and r is the value of x modulo n.

We can also define a function *trans*, which transposes pitches:

```
trans :: Int → Pitch → Pitch
trans i p = pitch (absPitch p + i)
```

Exercise 2.1 Show that *abspitch* (*pitch* *ap*) = *ap*, and, up to enharmonic equivalences, *pitch* (*abspitch* p) = p.

Exercise 2.2 Show that *trans* i (*trans* j p) = *trans* $(i + j)$ p.

Chapter 3

Polymorphic and Higher-Order Functions

In the last chapter we learned a little about polymorphic data types. In this chapter we will also learn about *polymorphic functions*, which are essentially functions defined over polymorphic data types. The already familiar *list* is the most common example of a polymorphic data type, and we will study it in depth in this chapter. Although lists have no direct musical connection, they are perhaps the most commonly used data type in Haskell, and have many applications in computer music programming.

We will also learn about *higher-order functions*, which are functions that take one or more functions as arguments or return a function as a result (functions can also be placed in data structures, making the data constructors higher-order too). Together, polymorphic and higher-order functions substantially increase our expressive power and our ability to reuse code. We will see that both of these new ideas naturally follow the foundations that we have already built.

(A more detailed discussion of pre-defined polymorphic functions that operate on lists can be found in Chapter A.)

3.1 Polymorphic Types

In previous chapters we saw examples of lists containing several different kinds of elements—integers, characters, pitch classes, and so on—and you can well imagine situations requiring lists of other element types as well. Sometimes, however, we don't wish to be so particular about the precise type of the elements. For example, suppose we want to define a function *length* that determines the number of elements in a list. We don't really care whether the list contains integers, pitch classes, or even other lists—we imagine computing the length in exactly the same way in each case. The obvious definition is:

$$\begin{array}{l}
\mathit{length}\ [\,] = 0 \\
\mathit{length}\ (x : \mathit{xs}) = 1 + \mathit{length}\ \mathit{xs}
\end{array}$$

This recursive definition is self-explanatory. We can read the equations as saying: "The length of the empty list is 0, and the length of a list whose first element is x and remainder is xs is 1 plus the length of xs."

But what should the type of *length* be? Intuitively, what we'd like to say is that, for *any* type a, the type of *length* is $[\,a\,] \rightarrow \mathit{Integer}$. In Haskell we write this simply as:

$$\mathit{length} :: [\,a\,] \rightarrow \mathit{Integer}$$

Details: Generic names for types, such as a above, are called *type variables*, and are uncapitalized to distinguish them from specific types such as $\mathit{Integer}$.

So *length* can be applied to a list containing elements of *any* type. For example:

$$\begin{array}{lcl}
\mathit{length}\ [1,2,3] & \Longrightarrow & 3 \\
\mathit{length}\ [\,C, \mathit{Cs}, \mathit{Df}\,] & \Longrightarrow & 3 \\
\mathit{length}\ [[1],[\,],[2,3,4]] & \Longrightarrow & 3
\end{array}$$

Note that the type of the argument to *length* in the last example is $[[\mathit{Integer}]]$; that is, a list of lists of integers.

Here are two other examples of polymorphic list functions, which happen to be pre-defined in Haskell:

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

Details: The _ on the left-hand side of these equations is called a *wildcard* pattern. It matches any value, and binds no variables. It is useful as a way of documenting the fact that we do not care about the value in that part of the pattern.

These two functions take the "head" and "tail," respectively, of any non-empty list:

$\mathit{head}\ [1,2,3] \Rightarrow 1$
$\mathit{head}\ [\texttt{'a'},\texttt{'b'},\texttt{'c'}] \Rightarrow \texttt{'a'}$
$\mathit{tail}\ [1,2,3] \Rightarrow [2,3]$
$\mathit{tail}\ [\texttt{'a'},\texttt{'b'},\texttt{'c'}] \Rightarrow [\texttt{'b'},\texttt{'c'}]$

Functions such as *length*, *head*, and *tail* are said to be *polymorphic* (*poly* means *many* and *morphic* refers to the structure, or *form*, of objects). Polymorphic functions arise naturally when defining functions on lists and other polymorphic data types, including the *Music* data type defined in the last chapter.

3.2 Abstraction Over Recursive Definitions

Suppose we have a list of pitches, and we wish to convert each of them to an absolute pitch. We might write a function:

$\mathit{toAbsPitches} :: [\mathit{Pitch}] \to [\mathit{AbsPitch}]$
$\mathit{toAbsPitches}\ [\,] = [\,]$
$\mathit{toAbsPitches}\ (p : \mathit{ps}) = \mathit{absPitch}\ p : \mathit{toAbsPitches}\ \mathit{ps}$

We might also want to convert a list of absolute pitches to a list of pitches:

$\mathit{toPitches} :: [\mathit{AbsPitch}] \to [\mathit{Pitch}]$
$\mathit{toPitches}\ [\,] = [\,]$
$\mathit{toPitches}\ (a : \mathit{as}) = \mathit{pitch}\ a : \mathit{toPitches}\ \mathit{as}$

These two functions are different, but share something in common: there is a repeating pattern of operations. But the pattern is not quite like any of the examples that we studied earlier, and therefore it is unclear how to apply the abstraction principle. What distinguishes this situation is that there is a repeating pattern of *recursion*.

In discerning the nature of a repeating pattern it's sometimes helpful to identify those things that *aren't* repeating—i.e. those things that are *changing*—since these will be the sources of *parameterization*: those values that must be passed as arguments to the abstracted function. In the case above, these changing values are the functions *absPitch* and *pitch*; let's consider them instances of a new name, f. If we then simply rewrite either of the above functions as a new function—let's call it *map*—that takes an extra argument f, we arrive at:

$$\begin{array}{l} \mathit{map}\ f\ [\,] = [\,] \\ \mathit{map}\ f\ (x : \mathit{xs}) = f\ x : \mathit{map}\ f\ \mathit{xs} \end{array}$$

With this definition of *map*, we can now redefine *toAbsPitches* and *toPitches* as:

$$\begin{array}{l} \mathit{toAbsPitches} :: [\mathit{Pitch}] \rightarrow [\mathit{AbsPitch}] \\ \mathit{toAbsPitches}\ \mathit{ps} = \mathit{map}\ \mathit{absPitch}\ \mathit{ps} \\ \\ \mathit{toPitches} :: [\mathit{AbsPitch}] \rightarrow [\mathit{Pitch}] \\ \mathit{toPitches}\ \mathit{as} = \mathit{map}\ \mathit{pitch}\ \mathit{as} \end{array}$$

Note that these definitions are non-recursive; the common pattern of recursion has been abstracted away and isolated in the definition of *map*. They are also very succinct; so much so, that it seems unnecessary to create new names for these functions at all! One of the powers of higher-order functions is that they permit concise yet easy-to-understand definitions such as this, and you will see many similar examples throughout the remainder of the text.

A proof that the new versions of these two functions are equivalent to the old ones can be done via calculation, but requires a proof technique called *induction*, because of the recursive nature of the original function definitions. We will discuss inductive proofs in detail, including these two examples, in Chapter 8.

3.2.1 Map is Polymorphic

What should the type of *map* be? Let's look first at its use in *toAbsPitches*: it takes the function $\mathit{absPitch} :: \mathit{Pitch} \rightarrow \mathit{AbsPitch}$ as its first argument, a list of *Pitch*s as its second argument, and it returns a list of *AbsPitch*s as its result. So its type must be:

$$\mathit{map} :: (\mathit{Pitch} \rightarrow \mathit{AbsPitch}) \rightarrow [\mathit{Pitch}] \rightarrow [\mathit{AbsPitch}]$$

Yet a similar analysis of its use in *toPitches* reveals that *map*'s type should be:

$$\mathit{map} :: (\mathit{AbsPitch} \rightarrow \mathit{Pitch}) \rightarrow [\mathit{AbsPitch}] \rightarrow [\mathit{Pitch}]$$

This apparent anomaly can be resolved by noting that *map*, like *length*, *head* and *tail*, does not really care what its list element types are, *as long as its functional argument can be applied to them.* Indeed, *map* is *polymorphic*, and its most general type is:

$$map :: (a \rightarrow b) \rightarrow [\,a\,] \rightarrow [\,b\,]$$

This can be read: "*map* is a function that takes a function from any type a to any type b, and a list of a's, and returns a list of b's." The correspondence between the two a's and between the two b's is important: a function that converts *Int*'s to *Char*'s, for example, cannot be mapped over a list of *Char*'s. It is easy to see that in the case of *toAbsPitches*, a is instantiated as *Pitch* and b as *AbsPitch*, whereas in *toPitches*, a and b are instantiated as *AbsPitch* and *Pitch*, respectively.

> **Details:** In Chapter 1 we mentioned that every expression in Haskell has an associated type. But with polymorphism, you might wonder if there is just one type for every expression. For example, *map* could have any of these types:
>
> $(a \rightarrow b) \rightarrow [\,a\,] \rightarrow [\,b\,]$
> $(\mathit{Integer} \rightarrow b) \rightarrow [\,\mathit{Integer}\,] \rightarrow [\,b\,]$
> $(a \rightarrow \mathit{Float}) \rightarrow [\,a\,] \rightarrow [\,\mathit{Float}\,]$
> $(\mathit{Char} \rightarrow \mathit{Char}) \rightarrow [\,\mathit{Char}\,] \rightarrow [\,\mathit{Char}\,]$
>
> and so on, depending on how it will be used. However, notice that the first of these types is in some fundamental sense more general than the other three. In fact, every expression in Haskell has a unique type known as its *principal type*: the least general type that captures all valid uses of the expression. The first type above is the principal type of *map*, since it captures all valid uses of *map*, yet is less general than, for example, the type $a \rightarrow b \rightarrow c$. As another example, the principal type of *head* is $[\,a\,] \rightarrow a$; the types $[\,b\,] \rightarrow a$, $b \rightarrow a$, or even a are too general, whereas something like $[\,\mathit{Integer}\,] \rightarrow \mathit{Integer}$ is too specific.[1]

3.2.2 Using map

Now that we can picture *map* as a polymorphic function, it is useful to look back on some of the examples we have worked through to see if there are any situations where *map* might have been useful. For example, recall from Section 1.4.3 the definition of *totalArea*:

$$\mathit{totalArea} = \mathit{listSum}\ [\,\mathit{circleArea}\ r1, \mathit{circleArea}\ r2, \mathit{circleArea}\ r3\,]$$

It should be clear that this can be rewritten as:

[1] The existence of unique principal types is the hallmark feature of the *Hindley-Milner type system* [Hin69, Mil78] that forms the basis of the type systems of Haskell, ML [MTH90] and many other functional languages [Hud89].

$$\mathit{totalArea} = \mathit{listSum}\ (\mathit{map}\ \mathit{circleArea}\ [\mathit{r1}, \mathit{r2}, \mathit{r3}])$$

A simple calculation is all that is needed to show that these are the same:

$$\begin{aligned}
&\mathit{map}\ \mathit{circleArea}\ [\mathit{r1}, \mathit{r2}, \mathit{r3}]\\
&\Rightarrow \mathit{circleArea}\ \mathit{r1} : \mathit{map}\ \mathit{circleArea}\ [\mathit{r2}, \mathit{r3}]\\
&\Rightarrow \mathit{circleArea}\ \mathit{r1} : \mathit{circleArea}\ \mathit{r2} : \mathit{map}\ \mathit{circleArea}\ [\mathit{r3}]\\
&\Rightarrow \mathit{circleArea}\ \mathit{r1} : \mathit{circleArea}\ \mathit{r2} : \mathit{circleArea}\ \mathit{r3} : \mathit{map}\ \mathit{circleArea}\ [\,]\\
&\Rightarrow \mathit{circleArea}\ \mathit{r1} : \mathit{circleArea}\ \mathit{r2} : \mathit{circleArea}\ \mathit{r3} : [\,]\\
&\Rightarrow [\mathit{circleArea}\ \mathit{r1}, \mathit{circleArea}\ \mathit{r2}, \mathit{circleArea}\ \mathit{r3}]
\end{aligned}$$

For an interesting musical example, let's generate a whole-tone scale starting at a given pitch:

```
wts :: Pitch → [Music Pitch]
wts p = let ap = absPitch p
            f ap = note qn (pitch ap)
        in map f [mc, mc + 2 .. mc + 12]
```

Details: A list $[a, b \mathinner{..} c]$ is called an *arithmetic sequence*, and is special syntax for the list $[a, a + d, a + 2 * d, ..., c]$ where $d = b - a$.

3.3 Append

Let's now consider the problem of *concatenating* or *appending* two lists together; that is, creating a third list that consists of all of the elements from the first list followed by all of the elements of the second. Once again the type of list elements does not matter, so we will define this as a polymorphic infix operator (++):

```
(++) :: [a] → [a] → [a]
```

For example, here are two uses of (++) on different types:

$$\begin{aligned}
&[1, 2, 3] +\!\!+ [4, 5, 6] \Longrightarrow [1, 2, 3, 4, 5, 6]\\
&[C, E, G] +\!\!+ [D, F, A] \Longrightarrow [C, E, G, D, F, A]
\end{aligned}$$

As usual, we can approach this problem by considering the various possibilities that could arise as input. But in the case of (++) we are given *two* inputs—so which do we consider first? In general this is not an easy question, but in the case of (++) we can get a hint about what to do by noting that the result contains firstly all of the elements from the first list. So let's consider the first list first: it could be empty, or non-empty. If it is empty the answer is easy:

$$[\,] +\!\!+ \; ys = ys$$

and if it is not empty the answer is also straightforward:

$$(x : xs) +\!\!+ \; ys = x : (xs +\!\!+ \; ys)$$

Note the recursive use of $(+\!\!+)$. Our full definition is thus:

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

The Efficiency and Fixity of Append In Chapter 8 we will prove the following simple property about $(+\!\!+)$:

$$(xs +\!\!+ \; ys) +\!\!+ \; zs = xs +\!\!+ \; (ys +\!\!+ \; zs)$$

That is, $(+\!\!+)$ is *associative*.

But what about the efficiency of the left-hand and right-hand sides of this equation? It is easy to see via calculation that appending two lists together takes a number of steps proportional to the length of the first list (indeed the second list is not evaluated at all). For example:

$$\begin{array}{l}
[1, 2, 3] +\!\!+ \; xs \\
\Rightarrow 1 : ([2, 3] +\!\!+ \; xs) \\
\Rightarrow 1 : 2 : ([3] +\!\!+ \; xs) \\
\Rightarrow 1 : 2 : 3 : ([\,] +\!\!+ \; xs) \\
\Rightarrow 1 : 2 : 3 : xs
\end{array}$$

Therefore the evaluation of $xs +\!\!+ \; (ys +\!\!+ \; zs)$ takes a number of steps proportional to the length of xs plus the length of ys. But what about $(xs +\!\!+ \; ys) +\!\!+ \; zs$? The leftmost append will take a number of steps proportional to the length of xs, but then the rightmost append will require a number of steps proportional to the length of xs plus the length of ys, for a total cost of:

$$2 * \mathit{length}\ xs + \mathit{length}\ ys$$

Thus $xs +\!\!+ \; (ys +\!\!+ \; zs)$ is more efficient than $(xs +\!\!+ \; ys) +\!\!+ \; zs$. This is why the Standard Prelude defines the fixity of $(+\!\!+)$ as:

$$\textbf{infixr}\ 5 +\!\!+$$

In other words, if you just write $xs +\!\!+ \; ys +\!\!+ \; zs$, you will get the most efficient association, namely the right association $xs +\!\!+ \; (ys +\!\!+ \; zs)$. In the next section we will see a more dramatic example of this property.

3.4 Fold

Suppose we wish to take a list of notes (each of type *Music*) and convert them into a *line*, or *melody*. We can define a recursive function to do this:

```
line :: [Music a] → Music a
line [ ] = rest 0
line (m : ms) = m :+: line ms
```

We might also wish to have a function *chord* that operates in an analogous way, but using (:=:) instead of (:+:):

```
chord :: [Music a] → Music a
chord [ ] = rest 0
chord (m : ms) = m :=: chord ms
```

In a completely different context we might wish to compute the highest pitch in a list of pitches:

```
maxPitch :: [Pitch] → Pitch
maxPitch [ ] = pitch 0
maxPitch (p : ps) = p !!! maxPitch ps
```

where !!! is defined as:

```
p1 !!! p2 = if absPitch p1 > absPitch p2 then p1 else p2
```

Details: An expression **if** *pred* **then** *cons* **else** *alt* is called a *conditional expression*. If *pred* (called the *predicate*) is true, then *cons* (called the *consequence*) is the result; if *pred* is false, then *alt* (called the *alternative*) is the result.

Once again we have a situation where several definitions share something in common—a repeating recursive pattern. Using the process that we used to discover *map*, let's first identify those things that are changing. There are two pairs: the *rest* 0 and *pitch* 0 values (for which we'll use the generic name *init*, for "initial value"), and the (:+:) and (!!!) operators (for which we'll use the generic name *op*, for "operator"). If we now rewrite either of the above functions as a new function—lets call it *fold*—that takes extra arguments *op* and *init*, we arrive at:[2]

[2] The use of the name "*fold*" for this function is historical, and has little to do with the use of "fold" and "unfold" to describe steps in a calculation.

```
fold op init [ ] = init
fold op init (x : xs) = x ‘op‘ fold op init xs
```

> **Details:** Any normal binary function name can be used as an infix operator by enclosing it in backquotes; x ‘f‘ y is equivalent to $f\ x\ y$. Using infix application here for *op* better reflects the structure of the repeating pattern that we are abstracting.

With this definition of *fold* we can now rewrite the definitions of *line*, *chord*, and *maxPitch* as:

```
line, chord :: [Music] → Music
line ms = fold (:+:) (rest 0) ms
chord ms = fold (:=:) (rest 0) ms

maxPitch :: [Pitch] → Pitch
maxPitch ps = fold (!!!) 0 ps
```

> **Details:** Just as we can turn a function into an operator by enclosing it in backquotes, we can turn an operator into a function by enclosing it in parentheses. This is required in order to pass an operator as a value to another function, as in the examples above. (If we wrote *fold* !!! 0 *ps* instead of *fold* (!!!) 0 *ps* it would look like we were trying to compare *fold* to 0 *ps*, which is nonsensical and ill-typed.)

In Chapter 8 we will use induction to prove that these new definitions are equivalent to the old ones.

As another example, recall the definition of *listSum* from Section 1.4.3:

```
listSum :: [Float] → Float
listSum [ ] = 0
listSum (x : xs) = x + listSum xs
```

We can now rewrite this more succinctly using *fold*:

```
listSum :: [Float] → Float
listSum xs = fold (+) 0 xs
```

fold, like *map*, is a highly useful—reusable—function, as we will see through several other examples later in the text. Indeed, it too is polymorphic, for note that it does not depend on the type of the list elements. Its most general type—somewhat trickier than that for *map*—is:

```
fold :: (a → b → b) → b → [a] → b
```

This allows us to use *fold* whenever we need to “collapse” a list of elements using a binary (i.e. two-argument) operator.

3.4.1 Haskell's Folds

Haskell actually defines two versions of *fold* in the Standard Prelude. The first is called *foldr* ("fold-from-the-right") which is defined the same as our *fold*:

$$\begin{array}{l} \mathit{foldr} :: (a \to b \to b) \to b \to [\,a\,] \to b \\ \mathit{foldr}\ \mathit{op}\ \mathit{init}\ [\,] = \mathit{init} \\ \mathit{foldr}\ \mathit{op}\ \mathit{init}\ (x : \mathit{xs}) = x\ `\mathit{op}`\ \mathit{foldr}\ \mathit{op}\ \mathit{init}\ \mathit{xs} \end{array}$$

A good way to think about *foldr* is that it replaces all occurences of the list operator (:) with its first argument (a function), and replaces [] with its second argument. In other words:

$$\begin{array}{l} \mathit{foldr}\ \mathit{op}\ \mathit{init}\ (\mathit{x1} : \mathit{x2} : ... : \mathit{xn} : [\,]) \\ \Longrightarrow \mathit{x1}\ `\mathit{op}`\ (\mathit{x2}\ `\mathit{op}`\ (...(\mathit{xn}\ `\mathit{op}`\ \mathit{init})...)) \end{array}$$

This might help you to understand the type of *foldr* better, and also explains its name: the list is "folded from the right." Stated another way, for any list *xs*, the following always holds:[3]

$$\mathit{foldr}\ (:)\ [\,]\ \mathit{xs} \Longrightarrow \mathit{xs}$$

Haskell's second version of *fold* is called *foldl*:

$$\begin{array}{l} \mathit{foldl} :: (b \to a \to b) \to b \to [\,a\,] \to b \\ \mathit{foldl}\ \mathit{op}\ \mathit{init}\ [\,] = \mathit{init} \\ \mathit{foldl}\ \mathit{op}\ \mathit{init}\ (x : \mathit{xs}) = \mathit{foldl}\ \mathit{op}\ (\mathit{init}\ `\mathit{op}`\ x)\ \mathit{xs} \end{array}$$

A good way to think about *foldl* is to imagine "folding the list from the left:"

$$\begin{array}{l} \mathit{foldl}\ \mathit{op}\ \mathit{init}\ (\mathit{x1} : \mathit{x2} : ... : \mathit{xn} : [\,]) \\ \Longrightarrow (...((\mathit{init}\ `\mathit{op}`\ \mathit{x1})\ `\mathit{op}`\ \mathit{x2})...)\ `\mathit{op}`\ \mathit{xn} \end{array}$$

3.4.2 Why Two Folds?

Note that if we had used *foldl* instead of *foldr* in the definitions given earlier then not much would change; *foldr* and *foldl* would give the same result. Indeed, judging from their types, it looks like the only difference between *foldr* and *foldl* is that the operator takes its arguments in a different order.

So why does Haskell define two versions of *fold*? It turns out that there are situations where using one is more efficient, and possibly "more defined,"

[3] We will formally prove this in Chapter 8.

than the other. (By more defined, we mean that the function terminates on more values of its input domain.)

Probably the simplest example of this is a generalization of the associativity of $(+\!\!+)$ discussed in the last section. Suppose that we wish to collapse a list of lists into one list. The Standard Prelude defines the polymorphic function *concat* for this purpose:

$$\begin{array}{l} \mathit{concat} :: [[a]] \rightarrow [a] \\ \mathit{concat}\ \mathit{xss} = \mathit{foldr}\ (+\!\!+)\ [\,]\ \mathit{xss} \end{array}$$

For example:

$$\begin{array}{l} \mathit{concat}\ [[1],[3,4],[\,],[5,6]] \\ \Rightarrow [1,2,3,4,5,6] \end{array}$$

More importantly, from the earlier discussion it should be clear that this property holds:

$$\begin{array}{l} \mathit{concat}\ [\mathit{xs1},\mathit{xs2},...,\mathit{xsn}] \\ \Rightarrow \mathit{foldr}\ (+\!\!+)\ [\,]\ [\mathit{xs1},\mathit{xs2},...,\mathit{xsn}] \\ \Rightarrow \mathit{xs1}\ +\!\!+\ (\mathit{xs2}\ +\!\!+\ (...(\mathit{xn}\ +\!\!+\ [\,]))...) \end{array}$$

The total cost of this computation is proportional to the sum of the lengths of all of the lists. If each list has the same length *len*, then this cost is $n * \mathit{len}$.

On the other hand, if we had defined *concat* this way:

$$\mathit{slowConcat}\ \mathit{xss} = \mathit{foldl}\ (+\!\!+)\ [\,]\ \mathit{xss}$$

then we have:

$$\begin{array}{l} \mathit{slowConcat}\ [\mathit{xs1},\mathit{xs2},...,\mathit{xsn}] \\ \Rightarrow \mathit{foldl}\ (+\!\!+)\ [\,]\ [\mathit{xs1},\mathit{xs2},...,\mathit{xsn}] \\ \Rightarrow (...(([\,]\ +\!\!+\ \mathit{x1})\ +\!\!+\ \mathit{x2})...)\ +\!\!+\ \mathit{xn} \end{array}$$

If each list has the same length *len*, then the cost of this computation will be:

$$\begin{array}{l} \mathit{len} + (\mathit{len} + \mathit{len}) + (\mathit{len} + \mathit{len} + \mathit{len}) + ... + (n-1) * \mathit{len} \\ \Rightarrow n * (n-1) * \mathit{len} \end{array}$$

which is considerably worse than $n * \mathit{len}$. Thus the choice of *foldr* in the definition of *concat* is quite important.

Similar examples can be given to demonstrate that *foldl* is sometimes more efficient than *foldr*. On the other hand, in many cases the choice does not matter at all (consider, for example, $(+)$). The moral of all this is that care must be taken in the choice between *foldr* and *foldl* if efficiency is a concern.

3.4.3 Fold for Non-empty Lists

One might argue that both *line* and *maxPitch* should not be well defined on an empty list, and for that purpose the Standard Prelude provides functions *foldr1* and *foldl1*, which return an error if applied to an empty list. In certain contexts this may in fact be the preferred behavior for *line* and *maxPitch*, as well as a function *chord* that is similar to *line* except that it does parallel composition. So we could define:

```
line1, chord1 :: [Music] → Music
line1 ms = foldr1 (:+:) ms
chord1 ms = foldr1 (:=:) ms

maxPitch1 :: [Pitch] → Pitch
maxPitch1 ps = foldr1 (!!!) ps
```

3.5 A Final Example: Reverse

As a final example of a useful list function, consider the problem of *reversing* a list, which we will capture in a function called *reverse*. For example, *reverse* $[1, 2, 3]$ is $[3, 2, 1]$. Thus *reverse* takes a single list argument, whose possibilities are the normal ones for a list: it is either empty, or it is not. And so we write:

```
reverse :: [a] → [a]
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]
```

This, in fact, is a perfectly good definition for *reverse*—it is certainly clear—except for one small problem: it is terribly inefficient! To see why, first note that the number of steps needed to compute *xs* ++ *ys* is proportional to the length of *xs*. Now suppose that the list argument to *reverse* has length n. The recursive call to *reverse* will return a list of length $n - 1$, which is the first argument to (++). Thus the cost to reverse a list of length of n will be proportional to $n - 1$ plus the cost to reverse a list of length $n - 1$. So the total cost is proportional to $(n - 1) + (n - 2) + \cdots + 1 = n(n - 1)/2$, which in turn is proportional to the square of n.

Can we do better than this? Yes we can.

There is another algorithm for reversing a list, which goes something like this: take the first element, and put it at the front of an empty auxiliary list; then take the next element and add it to the front of the auxiliary list (thus the auxiliary list now consists of the first two elements in the original list, but in reverse order); then do this again and again until you reach the end of the original list. At that point the auxiliary list will be the reverse of the original one.

This algorithm can be expressed recursively, but the auxiliary list implies that we need a function that takes *two* arguments—the original list and the auxiliary one—yet *reverse* only takes one. So we create an auxiliary function *rev*:

```
reverse xs = rev [] xs
  where rev acc [] = acc
        rev acc (x : xs) = rev (x : acc) xs
```

The auxiliary list is the first argument to *rev*, and is called *acc* since it behaves as an "accumulator" of the intermediate results. Note how it is returned as the final result once the end of the original list is reached.

A little thought should convince the reader that this function does not have the quadratic (n^2) behavior of the first algorithm, and indeed can be shown to execute a number of steps that is directly proportional to the length of the list, which we can hardly expect to improve upon.

But now, compare the definition of *rev* with the definition of *foldl*:

```
foldl op init [] = init
foldl op init (x : xs) = foldl op (init `op` x) xs
```

They are somewhat similar. In fact, suppose we were to slightly rewrite *rev*, yielding:

```
rev op acc [] = acc
rev op acc (x : xs) = rev op (acc `op` x) xs
```

Now *rev* looks exactly like *foldl*, and the question becomes whether or not there is a function that can be substituted for *op* that would make the latter definition of *rev* equivalent to the former one. Indeed there is:

```
revOp a b = b : a
```

For note that:

```
acc `revOp` x ⇒ revOp acc x ⇒ x : acc
```

So *reverse* can be rewritten as:

```
reverse xs = rev revOp [ ] xs
  where rev op acc [ ] = acc
        rev op acc (x : xs) = rev op (acc ‘op‘ x) xs
```

which is the same as:

```
reverse xs = foldl revOp [ ] xs
```

If all of this seems like magic, well, you are starting to see the beauty of functional programming!

3.6 Errors

In the last section we talked about the idea of “returning an error” when the argument to *foldr1* is the empty list. As you might imagine, there are other situations where an error result is also warranted.

There are many ways to deal with such situations, depending on the application, but sometimes we wish to literally stop the program, signalling to the user that some kind of an *error* has occurred. In Haskell this is done with the Standard Prelude function $error :: String \to a$. Note that *error* is polymorphic, meaning that it can be used with any data type. The value of the expression *error s* is $\perp$, the completely undefined, or “bottom” value. As an example of its use, here is the definition of *foldr1* from the Standard Prelude:

```
foldr1 :: (a → a → a) → [a] → a
foldr1 f [x] = x
foldr1 f (x : xs) = f x (foldr1 f xs)
foldr1 f [ ] = error "Prelude.foldr1: empty list"
```

Thus if the anomalous situation arises, the program will terminate immediately, and the string `"Prelude.foldr1: empty list"` will be printed.

> **Details:** Strings, i.e. sequences of characters, are written between double quotes in Haskell, as in `"Hello World"`. When typed on your computer, however, it will look a little differently, as in `"Hello World"` (the double-quote character is the same at both ends of the string). Strings have type *String*. The `"\n"` at the end of the string above is a “newline” character; that is, if another string were printed just after this one, it would appear beginning on the next line, rather than just after “Hello World.”

Exercise 3.1 What is the principal type of each of the following expressions:

map map
map foldl

Exercise 3.2 Rewrite the definition of *length* non-recursively.

Exercise 3.3 Define a function that behaves as each of the following:

1. Doubles each number in a list. For example:

   $$\mathit{doubleEach}\ [1,2,3] \Longrightarrow [2,4,6]$$

2. Pairs each element in a list with that number and one plus that number. For example:

   $$\mathit{pairAndOne}\ [1,2,3] \Longrightarrow [(1,2),(2,3),(3,4)]$$

3. Adds together each pair of numbers in a list. For example:

   $$\mathit{addEachPair}\ [(1,2),(3,4),(5,6)] \Longrightarrow [3,7,11]$$

In this exercise and the two that follow, give both recursive and (if possible) non-recursive definitions, and be sure to include type signatures.

Exercise 3.4 Define a function *maxList* that computes the maximum element of a list. Define *minList* analogously.

Exercise 3.5 Define a function that adds "pointwise" the elements of a list of pairs. For example:

$$\mathit{addPairsPointwise}\ [(1,2),(3,4),(5,6)] \Longrightarrow (9,12)$$

Exercise 3.6 Freddie the Frog wants to communicate privately with his girlfriend Francine by *encrypting* messages sent to her. Frog brains are not that large, so they agree on this simple strategy: each character in the text shall be converted to the character "one greater" than it, based on the representation described below (with wrap-around from 255 to 0). Define functions *encrypt* and *decrypt* that will allow Freddie and Francine to communicate using this strategy.

Hint: Characters are often represented inside a computer as some kind of an integer; in the case of Haskell, a 16-bit unicode representation is used. For this exercise, you will want to use two Haskell functions, *toEnum* and *fromEnum*. The first will convert an integer into a character, the second will convert a character into an integer.

Exercise 3.7 Suppose you are given a non-negative integer *amt* representing a sum of money, and a list of coin denominations $[v1, v2, ..., vn]$, each being a positive integer. Your job is to make change for *amt* using the coins in the coin supply. Define a function *makeChange* to solve this problem. For example, your function may behave like this:

$$makeChange\ 99\ [5,1] \Rightarrow [19,4]$$

where 99 is the amount and $[5,1]$ represents the types of coins (say, nickels and pennies in US currency) that we have. The answer $[19,4]$ means that we can make the exact change with 19 5-unit coins and 4 single-unit coins; this is the best (in terms of the total number of coins) possible solution.

To make things slightly easier, you may assume that the list representing the coin denominations is given in descending order, and that the single-unit coin is always one of the coin types.

[Need to add some musical exercises.]

Chapter 4

More About Higher-Order Functions

You have now seen several examples where functions are passed as arguments to other functions, such as with *fold* and *map*. In this chapter we will see several examples where functions are also returned as values. This will lead to several techniques for improving definitions that we have already written, techniques that we will use often in the remainder of the text.

4.1 Currying

The first improvement relates to the notation we have used to write function applications, such as *simple* $x\ y\ z$. Although we have seen the similarity of this to the mathematical notation *simple*(x, y, z), in fact there is an important difference, namely that *simple* $x\ y\ z$ is actually equivalent to $(((simple\ x)\ y)\ z)$. In other words, function application is *left associative*, taking one argument at a time.

Let's look at the expression $(((simple\ x)\ y)\ z)$ a bit closer: there is an application of *simple* to x, the result of which is applied to y; so $(simple\ x)$ must be a function! The result of this application, $((simple\ x)\ y)$, is then applied to z, so $((simple\ x)\ y)$ must also be a function!

Since each of these intermediate applications yields a function, it seems perfectly reasonable to define a function such as:

$$multSumByFive = simple\ 5$$

What is *simple* 5? From the above argument we know that it must be a function. And from the definition of *simple* in Section 1.1 we might guess that this function takes two arguments, and returns 5 times their sum. Indeed, we can *calculate* this result as follows:

$$\begin{array}{l}
\mathit{multSumByFive}\ a\ b \\
\Rightarrow (\mathit{simple}\ 5)\ a\ b \\
\Rightarrow \mathit{simple}\ 5\ a\ b \\
\Rightarrow 5 * (a + b)
\end{array}$$

The intermediate step with parentheses is included just for clarity. This method of applying functions to one argument at a time, yielding intermediate functions along the way, is called *currying*, after the logician Haskell B. Curry who popularized the idea.[1] It is helpful to look at the types of the intermediate functions as arguments are applied:

$$\begin{array}{l}
\mathit{simple} :: \mathit{Float} \to \mathit{Float} \to \mathit{Float} \to \mathit{Float} \\
\mathit{simple}\ 5 :: \mathit{Float} \to \mathit{Float} \to \mathit{Float} \\
\mathit{simple}\ 5\ a :: \mathit{Float} \to \mathit{Float} \\
\mathit{simple}\ 5\ a\ b :: \mathit{Float}
\end{array}$$

We use currying to improve some of our previous examples as follows. Suppose that I tell you that the expressions $f\ x$ and $g\ x$ are the same, for all values of x. Then it seems clear that the functions f and g are equivalent. So, if we want to define f in terms of g, instead of writing:

$$f\ x = g\ x$$

we can instead simply write:

$$f = g$$

Let's apply this reasoning to the definition of *line* from Section 3.4:

$$\mathit{line}\ \mathit{ms} = \mathit{fold}\ (:+:)\ (\mathit{Primitive}\ (\mathit{Rest}\ 0))\ \mathit{ms}$$

Since function application is left associative, we can rewrite this as:

$$\mathit{line}\ \mathit{ms} = (\mathit{fold}\ (:+:)\ (\mathit{Primitive}\ (\mathit{Rest}\ 0)))\ \mathit{ms}$$

But now applying the same reasoning here as we did for f and g means that we can write this simply as:

$$\mathit{line} = \mathit{fold}\ (:+:)\ (\mathit{Primitive}\ (\mathit{Rest}\ 0))$$

[1] It was actually Schönfinkel who first called attention to this idea [Sch24], but the word "schönfinkelling" is rather a mouthful!

Similarly, the definitions of *maxPitch* and *listSum*

$$\begin{array}{l} \mathit{maxPitch}\ \mathit{ps} = \mathit{fold}\ (!!!)\ 0\ \mathit{ps} \\ \mathit{listSum}\ \mathit{xs} = \mathit{foldl}\ (+)\ 0\ \mathit{xs} \end{array}$$

can be rewritten as:

$$\begin{array}{l} \mathit{maxPitch} = \mathit{fold}\ (!!!)\ 0 \\ \mathit{listSum} = \mathit{foldl}\ (+)\ 0 \end{array}$$

We will refer to this kind of simplification as "currying simplification" or just "currying," even though it actually has a more technical name, "eta contraction."

> **Details:** Some care should be taken when using this simplification idea. In particular, note that an equation such as $f\ x = g\ x\ y\ x$ cannot be simplified to $f = g\ x\ y$, since then the x would become undefined!

Here is a more interesting example, in which currying simplification is used three times. Recall from Section 3.5 the definition of *reverse* using *foldl*:

$$\begin{array}{l} \mathit{reverse}\ \mathit{xs} = \mathit{foldl}\ \mathit{revOp}\ [\,]\ \mathit{xs} \\ \quad \textbf{where}\ \mathit{revOp}\ \mathit{acc}\ x = x : \mathit{acc} \end{array}$$

Using the polymorphic function *flip* which is defined in the Standard Prelude as:

$$\begin{array}{l} \mathit{flip} :: (a \to b \to c) \to (b \to a \to c) \\ \mathit{flip}\ f\ x\ y = f\ y\ x \end{array}$$

it should be clear that *revOp* can be rewritten as:

$$\mathit{revOp}\ \mathit{acc}\ x = \mathit{flip}\ (:)\ \mathit{acc}\ x$$

But now currying simplification can be used twice to reveal that:

$$\mathit{revOp} = \mathit{flip}\ (:)$$

This, along with a third use of currying, allows us to rewrite the definition of *reverse* simply as:

$$\mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]$$

This is in fact the way *reverse* is defined in the Standard Prelude.

Exercise 4.1 Show that *flip* (*flip* f) is the same as f.

Exercise 4.2 What is the type of *ys* in:

$$\begin{aligned}&\mathit{xs} = [1,2,3] :: [\mathit{Float}]\\&\mathit{ys} = \mathit{map}\ (+)\ \mathit{xs}\end{aligned}$$

Exercise 4.3 Define a function *applyEach* that, given a list of functions, applies each to some given value. For example:

$$\mathit{applyEach}\ [\mathit{simple}\ 2\ 2, (+3)]\ 5 \Longrightarrow [14,8]$$

where *simple* is as defined in Section 1.1.

Exercise 4.4 Define a function *applyAll* that, given a list of functions $[\mathit{f1},\mathit{f2},...,\mathit{fn}]$ and a value v, returns the result $\mathit{f1}\ (\mathit{f2}\ (...(\mathit{fn}\ v)...))$. For example:

$$\mathit{applyAll}\ [\mathit{simple}\ 2\ 2, (+3)]\ 5 \Longrightarrow 20$$

Exercise 4.5 Recall the discussion about the efficiency of $(+\!\!+)$ and *concat* in Chapter 3. Which of the following functions is more efficient, and why?

$$\begin{aligned}&\mathit{appendr}, \mathit{appendl} :: [[a]] \rightarrow [a]\\&\mathit{appendr} = \mathit{foldr}\ (\mathit{flip}\ (+\!\!+))\ [\,]\\&\mathit{appendl} = \mathit{foldl}\ (\mathit{flip}\ (+\!\!+))\ [\,]\end{aligned}$$

4.2 Sections

With a bit more syntax, we can also curry applications of infix operators such as $(+)$. This syntax is called a *section*, and the idea is that, in an expression such as $(x+y)$, you can omit either the x or the y, and the result (with the parentheses still intact) is a function of that missing argument. If *both* variables are omitted, it is a function of *two* arguments. In other words, the expressions $(x+)$, $(+y)$ and $(+)$ are equivalent, respectively, to the functions:

$$\begin{aligned}&\mathit{f1}\ y = x + y\\&\mathit{f2}\ x = x + y\\&\mathit{f3}\ x\ y = x + y\end{aligned}$$

For example, suppose that we need to determine whether each number in a list is positive. Instead of writing:

```
posInts :: [Integer] → [Bool]
posInts xs = map test xs
             where test x = x > 0
```

we can simply write:

```
posInts :: [Integer] → [Bool]
posInts xs = map (>0) xs
```

which can be further simplified using currying:

```
posInts :: [Integer] → [Bool]
posInts = map (>0)
```

This is an extremely concise definition.

As you gain experience with higher-order functions you will not only be able to start writing definitions such as this directly, but you will also start *thinking* in "higher-order" terms. We will see many examples of this kind of reasoning throughout the text.

Exercise 4.6 Define a function *twice* that, given a function f, returns a function that applies f twice to its argument. For example:

```
(twice (+1)) 2 ⇒ 4
```

What is the principal type of *twice*? Describe what *twice twice* does, and give an example of its use. How about *twice twice twice* and *twice* (*twice twice*)?

Exercise 4.7 Generalize *twice* defined in the previous exercise by defining a function *power* that takes a function f and an integer n, and returns a function that applies the function f to its argument n times. For example:

```
power (+2) 5 1 ⟹ 11
```

Use *power* to define something (anything!) useful.

4.3 Anonymous Functions

The final way to define a function in Haskell is in some sense the most fundamental: it is called an *anonymous function*, or *lambda expressions* (since the concept is drawn directly from Church's lambda calculus [Chu41]). The idea is that functions are values, just like numbers and characters and strings, and therefore there should be a way to create them without having to give them a name. As a simple example, an anonymous function that increments its numeric argument by one can be written $\lambda x \to x + 1$. Anonymous functions are most useful in situations where you don't wish to name them, which is why they are called "anonymous."

> **Details:** The typesetting that we use in this book prints an actual Greek lambda character, but in writing $\lambda x \to x + 1$ in your programs you will have to write "`\x -> x+1`" instead.

As another example, to add one and then divide by two every element of a list, we could write:

$map\ (\lambda x \to (x + 1)\ /\ 2)\ xs$

An even better example is an anonymous function that pattern-matches its argument, as in:

$map\ (\lambda(a, b) \to a + b)\ xs$

> **Details:** Anonymous functions can only perform one match against an argument. That is, you cannot stack together several anonymous functions to define one function, as you can with equations.

Anonymous functions are considered most fundamental because definitions such as that for *simple* given in Chapter 1:

$simple\ x\ y\ z = x * (y + z)$

can be written instead as:

$simple = \lambda x\ y\ z \to x * (y + z)$

> **Details:** $\lambda x\ y\ z \to exp$ is shorthand for $\lambda x \to \lambda y \to \lambda z \to exp$.

We can also use anonymous functions to explain precisely the behavior of sections. In particular, note that:

$(x+) \Rightarrow \lambda y \to x + y$
$(+y) \Rightarrow \lambda x \to x + y$
$(+) \Rightarrow \lambda x\ y \to x + y$

Exercise 4.8 Suppose we define a function *fix* as:

$fix\ f = f\ (fix\ f)$

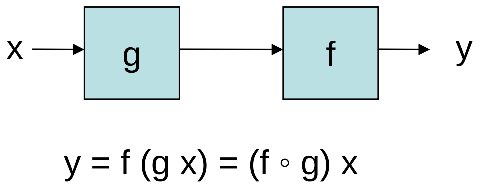


Figure 4.1: Gluing Two Functions Together

What is the principal type of *fix*? (This is tricky!) Suppose further that we have a recursive function:

```
remainder :: Integer → Integer → Integer
remainder a b = if a < b then a
                else remainder (a − b) b
```

Rewrite this function using *fix* so that it is not recursive. (Also tricky!) Do you think that this process can be applied to *any* recursive function?

4.4 Function Composition

An example of polymorphism that has nothing to do with data structures arises from the desire to take two functions f and g and “glue them together,” yielding another function h that first applies g to its argument, and then applies f to that result. This is called function *composition*, and Haskell pre-defines a simple infix operator ($\circ$) to achieve it, as follows:

```
(∘) :: (b → c) → (a → b) → a → c
(f ∘ g) x = f (g x)
```

> **Details:** The symbol for function composition is typeset in this book as ∘, which is the proper mathematical convention. When writing your programs, however, you will have to use a “period” , as in “`f . g`”.

Note the type of the operator ($\circ$); it is completely polymorphic. Note also that the result of the first function to be applied—some type b—must be the same as the type of the argument to the second function to be applied. Pictorially, if you think of a function as a black box that takes input at one end and returns some output at the other, function composition is like connecting two boxes together, end to end, as shown in Figure 4.1.

The ability to compose functions using ($\circ$) is extremely useful. For example, consider this function to compute the sum of the areas of circles with various radii:

$$\begin{array}{l} \mathit{totalCircleArea} :: [\mathit{Float}] \to \mathit{Float} \\ \mathit{totalCircleArea}\ \mathit{radii} = \mathit{listSum}\ (\mathit{map}\ \mathit{circleArea}\ \mathit{radii}) \end{array}$$

We can first add parentheses to emphasize the application of interest:

$$\begin{array}{l} \mathit{totalCircleArea} :: [\mathit{Float}] \to \mathit{Float} \\ \mathit{totalCircleArea}\ \mathit{radii} = \mathit{listSum}\ ((\mathit{map}\ \mathit{circleArea})\ \mathit{radii}) \end{array}$$

then rewrite as a function composition:

$$\begin{array}{l} \mathit{totalCircleArea} :: [\mathit{Float}] \to \mathit{Float} \\ \mathit{totalCircleArea}\ \mathit{radii} = (\mathit{listSum} \circ (\mathit{map}\ \mathit{circleArea}))\ \mathit{radii} \end{array}$$

and finally use currying to simplify:

$$\begin{array}{l} \mathit{totalCircleArea} :: [\mathit{Float}] \to \mathit{Float} \\ \mathit{totalCircleArea} = \mathit{listSum} \circ \mathit{map}\ \mathit{circleArea} \end{array}$$

Similarly, this definition:

$$\begin{array}{l} \mathit{totalSquareArea} :: [\mathit{Float}] \to \mathit{Float} \\ \mathit{totalSquareArea}\ \mathit{sides} = \mathit{listSum}\ (\mathit{map}\ \mathit{squareArea}\ \mathit{sides}) \end{array}$$

can be rewritten as:

$$\begin{array}{l} \mathit{totalSquareArea} :: [\mathit{Float}] \to \mathit{Float} \\ \mathit{totalSquareArea} = \mathit{listSum} \circ \mathit{map}\ \mathit{squareArea} \end{array}$$

But let's also create additional compositions. A function that determines whether all of the elements in a list are greater than zero, and one that determines if at least one is greater than zero, can be written:

$$\begin{array}{l} \mathit{allOverZero}, \mathit{oneOverZero} :: [\mathit{Integer}] \to \mathit{Bool} \\ \mathit{allOverZero} = \mathit{and} \circ \mathit{posInts} \\ \mathit{oneOverZero} = \mathit{or} \circ \mathit{posInts} \end{array}$$

Note that the auxiliary function *posInts* is simple enough that we could incorporate its definition directly, as in:

$$\begin{array}{l} \mathit{allOverZero}, \mathit{oneOverZero} :: [\mathit{Integer}] \to \mathit{Bool} \\ \mathit{allOverZero} = \mathit{and} \circ \mathit{map}\ ({>}0) \\ \mathit{oneOverZero} = \mathit{or} \circ \mathit{map}\ ({>}0) \end{array}$$

> **Details:** $\mathit{and} :: [\mathit{Bool}] \to \mathit{Bool}$ and $\mathit{or} :: [\mathit{Bool}] \to \mathit{Bool}$ are predefined functions that "and" and "or" together all of the elements in a list, returning a single Boolean result. The *Bool* type is predefined in Haskell simply as:

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

In the remainder of this text we will not refrain from writing definitions such as this directly, using a small set of rich polymorphic functions such as *fold* and *map*, plus a few others drawn from the Prelude and Standard Libraries.

Exercise 4.9 Rewrite this example:

$\mathit{map}\ (\lambda x \rightarrow (x + 1)\ /\ 2)\ \mathit{xs}$

using a composition of sections.

Exercise 4.10 Consider the expression:

$\mathit{map}\ f\ (\mathit{map}\ g\ \mathit{xs})$

Rewrite this using function composition and a single call to *map*. Then rewrite the earlier example:

$\mathit{map}\ (\lambda x \rightarrow (x + 1)\ /\ 2)\ \mathit{xs}$

as a "map of a map."

Exercise 4.11 Go back to any exercises prior to this chapter, and simplify your solutions using ideas learned here.

Exercise 4.12 Using higher-order functions that we have now defined, fill in the two missing functions, *f1* and *f2*, in the evaluation below so that it is valid:

$\mathit{f1}\ (\mathit{f2}\ (*)\ [1, 2, 3, 4])\ 5 \Rightarrow [5, 10, 15, 20]$

Chapter 5

More Music

```
module Haskore.MoreMusic where
import Haskore.Music
```

In this chapter we will explore a number of simple musical ideas, and contribute to a growing collection of Haskell functions for expressing those ideas.

5.1 Delay and Repeat

Suppose that we wish to describe a melody *m* accompanied by an identical voice a perfect 5th higher. In Haskore we can simply write *m* :=: *transpose* 7 *m*. Similarly, a canon-like structure involving *m* can be expressed as *m* :=: *delay d m*, where:

```
delay :: Dur → Music a → Music a
delay d m = rest d :+: m
```

More interestingly, Haskell's non-strict semantics also allows us to define *infinite* musical values. For example, a musical value may be repeated *ad nauseum* using this simple function:

```
repeatM :: Music a → Music a
repeatM m = m :+: repeatM m
```

Thus, for example, an infinite ostinato can be expressed in this way, and then used in different contexts that automatically extract only the portion that is actually needed. We will see more examples of this shortly.

Exercise 5.1 Define a function *repM* :: *Int* → *Music a* → *Music a* such that *repM n m* repeats *m n* times.

5.2 Inversion and Retrograde

The notions of inversion, retrograde, retrograde inversion, etc. as used in twelve-tone theory are also easily captured in Haskore. These terms are usually applied only to "lines" of notes, i.e. a melody (in twelve-tone theory it is called a "row"). The *retrograde* of a line is simply its reverse—i.e. the notes played in the reverse order. The *inversion* of a line is with respect to a given pitch (by convention usually the first pitch), where the intervals between successive pitches are inverted, i.e. negated. If the absolute pitch of the first note is *ap*, then each pich *p* is converted into an absolute pitch $(ap - absPitch\ p) + ap$, in other words $2 * ap - absPitch\ p$.

To do all this in Haskell, let's first define a transformation from a line created by *line* to a list:

```
lineToList :: Music a → [Music a]
lineToList n@(Primitive (Rest 0)) = [ ]
lineToList (n :+: ns) = n : lineToList ns
lineToList _ = error "lineToList: argument not created by line"
```

Using this function it is then straightforward to define *invert*, from which the other functions are easily defined via composition:

```
retro, invert, retroInvert, invertRetro :: Music Pitch → Music Pitch
invert m = line (map inv l)
  where l@(Primitive (Note _ r) : _) = lineToList m
            inv (Primitive (Note d p)) =
                           note d (pitch (2 ∗ absPitch r − absPitch p))
            inv (Primitive (Rest d)) = rest d
retro = line ∘ reverse ∘ lineToList
retroInvert = retro ∘ invert
invertRetro = invert ∘ retro
```

Exercise 5.2 Show that *retro* ∘ *retro*, *invert* ∘ *invert*, and *retroInvert* ∘ *invertRetro* are the identity on values created by *line*.

5.3 Polyrhythms

For some rhythmical ideas, first note that if *m* is a line of three eighth notes, then *tempo* (3 / 2) *m* is a *triplet* of eighth notes. In fact *tempo* can be used to create quite complex rhythmical patterns. For example, consider the

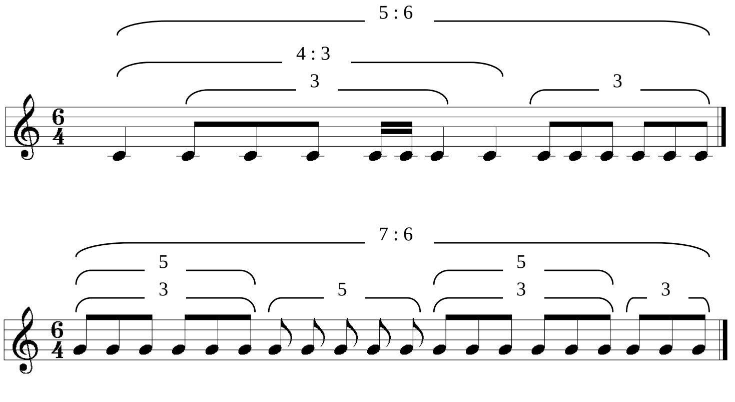


Figure 5.1: Nested Polyrhythms (top: *pr1*; bottom: *pr2*)

"nested polyrhythms" shown in Figure 5.1. They can be expressed naturally in Haskore as follows (note the use of the *where* clause in *pr2* to capture recurring phrases):

```
pr1, pr2 :: Pitch → Music Pitch
pr1 p = tempo (5 / 6)
            (tempo (4 / 3) (mkLn 1 p qn:+:
                            tempo (3 / 2) (mkLn 3 p en:+:
                                           mkLn 2 p sn:+:
                                           mkLn 1 p qn):+:
                            mkLn 1 p qn):+:
             tempo (3 / 2) (mkLn 6 p en))

pr2 p = tempo (7 / 6) (m1:+:
                       tempo (5 / 4) (mkLn 5 p en):+:
                       m1:+:
                       tempo (3 / 2) m2)
        where m1 = tempo (5 / 4) (tempo (3 / 2) m2 :+: m2)
              m2 = mkLn 3 p en

mkLn n p d = line (take n (repeat (note d p)))
```

Details: *take n lst* is the first n elements of the list *lst*. For example, *take* $3\ [C, Cs, Df, D, Ds] \Longrightarrow [C, Cs, Df]$. *repeat* x is the infinite list of the same value x. For example, *take* 3 (*repeat* 42) $\Longrightarrow [42, 42, 42]$.

To play polyrhythms *pr1* and *pr2* in parallel using middle C and middle G, respectively, we do the following:

```
pr12 :: Music Pitch
pr12 = pr1 (C, 4) :=: pr2 (G, 4)
```

5.4 Symbolic Meter Changes

We can implement the notion of “symbolic meter changes” of the form “oldnote = newnote” (quarter note = dotted eighth, for example) by defining an infix function:

```
(=:=) :: Dur → Dur → Music a → Music a
old =:= new = tempo (new / old)
```

Of course, using the new function is not much longer than using *Tempo* directly, but it may have nemonic value.

5.5 Computing Duration

It is often desirable to compute the *duration*, in whole notes, of a musical value; we can do so as follows:

```
dur :: Music a → Dur
dur (Primitive (Note d _)) = d
dur (Primitive (Rest d)) = d
dur (m1 :+: m2) = dur m1 + dur m2
dur (m1 :=: m2) = dur m1 `max` dur m2
dur (Modify (Tempo r) m) = dur m / r
dur (Modify _ m) = dur m
```

The duration of a primitive value is obvious. The duration of *m1* :+: *m2* is the sum of the two, and the duration of *m1* :=: *m2* is the maximum of the two. The only tricky part is the duration of a music value that is modified by the *Tempo* atttribute—in this case the duration must be scaled appropriately.

5.6 Super-retrograde

Using *dur* we can define a function *revM* that reverses any *Music* value (and is thus considerably more useful than *retro* defined earlier). Note the tricky treatment of (:=:).

```
revM :: Music a → Music a
revM n@(Primitive _) = n
revM (Modify c m) = Modify c (revM m)
revM (m1 :+: m2) = revM m2 :+: revM m1
revM (m1 :=: m2) =
  let d1 = dur m1
      d2 = dur m2
  in if d1 > d2 then revM m1 :=: (rest (d1 − d2) :+: revM m2)
                else (rest (d2 − d1) :+: revM m1) :=: revM m2
```

5.7 Truncating Parallel Composition

Note that the duration of *m1* :=: *m2* is the maximum of the durations of *m1* and *m2* (and thus if one is infinite, so is the result). Sometimes we would rather have the result be of duration equal to the shorter of the two. This is not as easy as it sounds, since it may require interrupting the longer one in the middle of a note (or notes).

We will define a "truncating parallel composition" operator (/=:), but first we will define an auxiliary function *cut* such that *cut d m* is the musical value *m* "cut short" to have at most duration *d*:

```
cut :: Dur → Music a → Music a
cut newDur m | newDur ⩽ 0 = rest 0
cut newDur (Primitive (Note oldDur x)) = note (min oldDur newDur) x
cut newDur (Primitive (Rest oldDur)) = rest (min oldDur newDur)
cut newDur (m1 :=: m2) = cut newDur m1 :=: cut newDur m2
cut newDur (m1 :+: m2) = let m1′ = cut newDur m1
                             m2′ = cut (newDur − dur m1′) m2
                         in m1′ :+: m2′
cut newDur (Modify (Tempo r) m) = tempo r (cut (newDur ∗ r) m)
cut newDur (Modify c m) = Modify c (cut newDur m)
```

Note that *cut* is equipped to handle a *Music* value of infinite length.

With *cut*, the definition of (/=:) is now straightforward:

```
(/=:) :: Music a → Music a → Music a
m1 /=: m2 = cut (min (dur m1) (dur m2)) (m1 :=: m2)
```

Unfortunately, whereas *cut* can handle infinite-duration music values, (/=:) cannot.

Exercise 5.3 Define a version of (/=:) that shortens correctly when either or both of its arguments are infinite in duration.

5.8 Trills

A *trill* is an ornament that alternates rapidly between two (usually adjacent) pitches. We will define two versions of a trill function, both of which take the starting note and an interval for the trill note as arguments (the interval is usually one or two, but can actually be anything). One version will additionally have an argument that specifies how long each trill note should be, whereas the other will have an argument that specifies how many trills should occur. In both cases the total duration will be the same as the duration of the original note.

Here is the first trill function:

```
trill :: Int → Dur → Music Pitch → Music Pitch
trill i sDur (Primitive (Note tDur p)) =
    if sDur ⩾ tDur then note tDur p
    else note sDur p
        :+: trill (negate i) sDur (note (tDur − sDur) (trans i p))
trill i d (Modify (Tempo r) m) = tempo r (trill i (d ∗ r) m)
trill i d (Modify c m) = Modify c (trill i d m)
trill _ _ _ = error "trill: input must be a single note."
```

It is simple to define a version of this function that starts on the trill note rather than the start note:

```
trill′ :: Int → Dur → Music Pitch → Music Pitch
trill′ i sDur m = trill (negate i) sDur (transpose i m)
```

The second way to define a trill is in terms of the number of subdivided notes to be included in the trill. We can use the first trill function to define this new one:

```
trilln :: Int → Int → Music Pitch → Music Pitch
trilln i nTimes m = trill i (dur m / fromIntegral nTimes) m
```

This, too, can be made to start on the other note.

```
trilln′ :: Int → Int → Music Pitch → Music Pitch
trilln′ i nTimes m = trilln (negate i) nTimes (transpose i m)
```

Finally, a *roll* can be implemented as a trill whose interval is zero. This feature is particularly useful for percussion.

```
roll :: Dur → Music Pitch → Music Pitch
rolln :: Int → Music Pitch → Music Pitch

roll dur m = trill 0 dur m
rolln nTimes m = trilln 0 nTimes m
```

5.9 Percussion

Percussion is a difficult notion to represent in the abstract. On one hand, a percussion instrument is just another instrument, so why should it be treated differently? On the other hand, even common practice notation treats it specially, even though it has much in common with non-percussive notation. The MIDI standard is equally ambiguous about the treatment of percussion: on one hand, percussion sounds are chosen by specifying an octave and pitch, just like any other instrument; on the other hand these pitches have no tonal meaning whatsoever: they are just a convenient way to select from a large number of percussion sounds. Indeed, part of the General MIDI Standard is a set of names for commonly used percussion sounds.

Since MIDI is such a popular platform, we can at least define some handy functions for using the General MIDI Standard. We start by defining the data type shown in Figure 5.2, which borrows its constructor names from the General MIDI standard. The comments reflecting the "MIDI Key" numbers will be explained later, but basically a MIDI Key is the equivalent of an absolute pitch in Haskore terminology. So all we need is a way to convert these percussion sound names into a *Music* value; i.e. a *Note*:

```
perc :: PercussionSound → Dur → Music Pitch
perc ps dur = note dur (pitch (fromEnum ps + 35))
```

Details: *fromEnum* is a method in the *Enum* class, which is all about enumerations. A data type that is a member of this class can be *enumerated*—i.e. the elements of the data type can be listed in order. *fromEnum* maps each value to its index in this enumeration. Thus *fromEnum AcousticBassDrum* is 0, *fromEnum BassDrum1* is 1, and so on.

For example, here are eight bars of a simple rock or "funk groove" that uses *perc* and *roll*:

```
data PercussionSound =
         AcousticBassDrum     -- MIDI Key 35
       | BassDrum1     -- MIDI Key 36
       | SideStick     -- ...
       | AcousticSnare | HandClap | ElectricSnare | LowFloorTom
       | ClosedHiHat | HighFloorTom | PedalHiHat | LowTom
       | OpenHiHat | LowMidTom | HiMidTom | CrashCymbal1
       | HighTom | RideCymbal1 | ChineseCymbal | RideBell
       | Tambourine | SplashCymbal | Cowbell | CrashCymbal2
       | Vibraslap | RideCymbal2 | HiBongo | LowBongo
       | MuteHiConga | OpenHiConga | LowConga | HighTimbale
       | LowTimbale | HighAgogo | LowAgogo | Cabasa
       | Maracas | ShortWhistle | LongWhistle | ShortGuiro
       | LongGuiro | Claves | HiWoodBlock | LowWoodBlock
       | MuteCuica | OpenCuica | MuteTriangle
       | OpenTriangle     -- MIDI Key 82
   deriving (Show, Eq, Ord, Enum)
```

Figure 5.2: General MIDI Percussion Names

```
funkGroove
   = let p1 = perc LowTom qn
         p2 = perc AcousticSnare en
     in tempo 3 (instrument Percussion (cut 8 (repeatM
          ((p1 :+: qnr :+: p2 :+: qnr :+: p2:+:
            p1 :+: p1 :+: qnr :+: p2 :+: enr)
            :=: roll en (perc ClosedHiHat 2))
        )))
```

Exercise 5.4 Define a function $chrom :: Pitch \to Pitch \to Music\ Pitch$ such that *chrom p1 p2* is a chromatic scale of quarter-notes whose first pitch is *p1* and last pitch is *p2*. If $p1 > p2$, the scale should be descending, otherwise it should be ascending. If $p1 == p2$, then the scale should contain just one note. (A chromatic scale is one whose successive pitches are separated by one absolute pitch, i.e. one semitone).

Exercise 5.5 Abstractly, a scale can be described by the intervals between successive notes. For example, the 8-note major scale can be defined as the sequence of 7 intervals $[2, 2, 1, 2, 2, 2, 1]$, and the 12-note chromatic scale by the 11 intervals $[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$. Define a function $mkScale :: Pitch \to [Int] \to Music\ Pitch$ such that *mkScale p* ints is the scale beginning at pitch *p* and having the intervallic structure *ints*.

Exercise 5.6 Write the melody of ”Frere Jacques” (or, ”Are You Sleeping”) in Haskore. Try to make it as succinct as possible. Then, using functions already defined, generate a four-part round, i.e. four identical voices, each delayed successively by two measures. Use a different instrument to realize each voice.

5.10 A Map for Music

Recall from Chapter 3 the definition of *map*:

```
map :: (a → b) → [a] → [b]
map f [ ] = [ ]
map f (x : xs) = f x : map f xs
```

This function is defined on the list data type. Is there something analogous for *Music*? I.e. a function:

```
mMap :: (a → b) → Music a → Music b
```

Such a function is indeed straightforward to define, but it helps to first define a map-like function for the *Primitive* type:

```
pMap :: (a → b) → Primitive a → Primitive b
pMap f (Note d x) = Note d (f x)
pMap f (Rest d) = Rest d
```

With *pMap* in hand we can now define *mMap*:

```
mMap :: (a → b) → Music a → Music b
mMap f (Primitive x) = Primitive (pMap f x)
mMap f (x :+: y) = mMap f x :+: mMap f y
mMap f (x :=: y) = mMap f x :=: mMap f y
mMap f (Modify c x) = Modify c (mMap f x)
```

Just as *map f xs* for lists replaces each polymorphic element x in xs with $f\ x$, *mMap f m* for *Music* replaces each polymorphic element p in m with $f\ p$.

As an example of how *mMap* can be used, suppose that we introduce a *Volume* type for a note simply as:

```
type Volume = Integer
```

and then wish to convert a value of type *Music Pitch* to a value of type *Music* (*Pitch*, *Volume*) – that is, we wish to pair each pitch with a volume attribute. We can define a function to do so as follows:

```
addVolume :: Volume → Music Pitch → Music (Pitch, Volume)
addVolume v = mMap (λp → (p, v))
```

Exercise 5.7 Using *mMap*, define a function:

```
scaleVolume :: Rational → Music (Pitch, Volume) → Music (Pitch, Volume)
```

such that *scaleVolume s m* scales the volume of each note in *m* by a factor of *s*.

5.11 A Fold for Music

We can also define a fold-like operator for *Music*. But whereas the list data type has only two constructors (the nullary constructor [ ] and the binary constructor (:)), *Music* has *four* constructors, and thus we define:

```
mFold :: (b → b → b) → (b → b → b) → (Primitive a → b)
        → (Control → b → b) → Music a → b
mFold (+ :) (=:) f g m =
  let rec = mFold (+ :) (=:) f g
  in case m of
         Primitive p → f p
         m1 :+: m2 → rec m1 + : rec m2
         m1 :=: m2 → rec m1 =: rec m2
         Modify c m → g c (rec m)
```

This somewhat unwieldy function basically takes apart a *Music* value and puts it back together with different constructors. Indeed, note that:

```
mFold (:+:) (:=:) Primitive Modify == id
```

Exercise 5.8 Prove the above property.

To see how *mFold* might be used, note first of all that it is more general than *mMap*—indeed, *mMap* can be defined in terms of *mFold* like this:

```
mMap′ :: (a → b) → Music a → Music b
mMap′ f = mFold (:+:) (:=:) g Modify where
  g (Note d x) = Primitive (Note d (f x))
  g (Rest d) = Primitive (Rest d)
```

More interestingly, we can use *mFold* to redefine things like the *dur* function from Section 5.5:

```
dur′ :: Music a → Dur
dur′ = mFold (+) max getDur modDur where
  getDur (Note d _) = d
  getDur (Rest d) = d
  modDur (Tempo r) d = d / r
  modDur _ d = d
```

Exercise 5.9 Redefine *revM* from Section 5.6 using *mFold*.

Exercise 5.10 Define a function *insideOut* that inverts the role of serial and parallel composition in a *Music* value. Using *insideOut*, see if you can, (a) find a non-trivial value *m* :: *Music Pitch* such that *m* == *insideOut m*, (b) find a value *m* :: *Music Pitch* such that:

```
m :+: insideOut m :+: m
```

sounds interesting. (You are free to define what "sounds interesting" means.)

Exercise 5.11 Find a simple piece of music written by your favorite composer, and transcribe it into Haskore. In doing so, look for repeating patterns, transposed phrases, etc. and reflect this in your code, thus revealing deeper structural aspects of the music than that found in common practice notation.

5.12 Crazy Recursion

With all the functions and data types that we have defined, and the power of recursion and higher-order functions at our finger tips, we can start to do some wild and crazy things. Here is just one such idea.

Let's define a function to recursively apply transformations *f* (to elements in a sequence) and *g* (to accumulated phrases) some specified number of times:

```
rep :: (Music a → Music a) → (Music a → Music a) → Int
    → Music a → Music a
rep f g 0 m = rest 0
rep f g n m = m :=: g (rep f g (n − 1) (f m))
```

With this simple function we can create some interesting phrases of music with very little code. For example, let's use *rep* three times, nested together, to create a "cascade" of sounds.

```
run = rep (transpose 5) (delay tn) 8 (c 4 tn)
cascade = rep (transpose 4) (delay en) 8 run
cascades = rep id (delay sn) 2 cascade
```

and then make the cascade run up, and then down:

```
final = cascades :+: revM cascades
```

What happens if we reverse the f and g arguments?

```
run' = rep (delay tn) (transpose 5) 8 (c 4 tn)
cascade' = rep (delay en) (transpose 4) 8 run'
cascades' = rep (delay sn) id 2 cascade'
final' = cascades' :+: revM cascades'
```

Exercise 5.12 Do something wild and crazy with Haskore.

Chapter 6

Interpretation and Performance

```
module Haskore.Performance
          where

import Haskore.Music
import Haskore.MoreMusic

instance Show (a → b) where
    showsPrec p f = showString "<<function>>"
```

6.1 Abstract Performance

So far, our presentation of musical values in Haskell has been entirely structural, i.e. *syntactic*. But what do these musical values actually *mean*, i.e. what is their *semantics*, or *interpretation*? The formal process of giving a semantic interpretation to syntactic constructs is very common in computer science, especially in programming language theory. But it is obviously also common in music: the interpretation of music is the very essence of musical performance. However, in conventional music this process is usually informal, appealing to aesthetic judgments and values. What we would like to do is make the process formal in Haskore—but still flexible, so that more than one interpretation is possible, just as in music.

To begin, we need to say exactly what an abstract *performance* is. Our approach is to consider a performance to be a time-ordered sequence of musical *events*, where each event captures the playing of one individual note. In Haskellese:

```
type Performance = [Event]

data Event = Event{ eTime :: PTime, eInst :: InstrumentName, ePitch :: AbsPitch,
                    eDur :: DurT, eVol :: Volume, pFields :: [Float]}
     deriving (Eq, Ord, Show)

type PTime = Rational
type DurT = Rational
type Volume = Integer
```

An event *Event*{ *eTime* = *s*, *eInst* = *i*, *ePitch* = *p*, *eDur* = *d*, *eVol* = *v*} captures the fact that at start time *s*, instrument *i* sounds pitch *p* with volume *v* for a duration *d* (where now duration is measured in seconds, rather than beats). (The *pField* of an event is for special instruments that require extra parameters, and will not be discussed much further in this chapter.)

An abstract performance is the lowest of our music representations not yet committed to MIDI, csound, or some other low-level computer music representation. In Chapter **??** we will discuss how to map a performance into MIDI.

Details: The data declaration for *Event* uses Haskell's *field label* syntax, also called *record* syntax, and is equivalent to:

```
data Event = Event PTime InstrumentName AbsPitch DurT Volume [Float]
     deriving (Eq, Ord, Show)
```

except that the former also defines "field labels" *eTime*, *eInst*, *ePitch*, *eDur*, *eVol*, and *pFields*, which can be used both to create and select from *Event* values. For example, this equation:

```
e = Event 0 Cello 27 (1 / 4) 50 []
```

is equivalent to:

```
e = Event{ eTime = 0, ePitch = 27, eDur = 1 / 4,
           eInst = Cello, eVol = 50, pFields = []}
```

The latter is more descriptive, however, and the order of the fields does not matter (indeed the order here is not the same as above).

Field labels can be used to select fields from an *Event* value; for example, *eInst e* ⇒ *Cello*, *eDur e* ⇒ 1 / 4, and so on. They can also be used to selectively *update* fields of an existing *Event* value. For example:

$e\{\,eInst = Flute\,\} \Rightarrow Event\ 0\ Flute\ 27\ (1\ /\ 4)\ 50\ [\,]$

Finally, they can be used selectively in pattern matching:

$f\ (Event\{\,eDur = d, ePitch = p\,\}) = ...d\ ...\ p...$

Field labels do not change the basic nature of a data type; they are simply a convenient syntax for referring to the components of a data type by name rather than by position.

To generate a complete performance of, i.e. give an interpretation to, a musical value, we must know the time to begin the performance, and the proper instrument, volume, key and tempo. In addition, to give flexibility to our interpretations, we must also know what *player* to use; that is, we need a mapping from the *PlayerName*s in a *Music* value to the actual players to be used.[1] We capture these ideas in Haskell as a "context" and "player map," respectively:

```
data Context a = Context{ cTime :: PTime, cPlayer :: Player a,
                           cInst :: InstrumentName, cDur :: DurT,
                           cKey :: Key, cVol :: Volume}
     deriving Show
type PMap a = PlayerName → Player a
type Key = AbsPitch
```

Finally, we are ready to give an interpretation to a piece of music, which we do by defining a function *perform*, which is conceptually perhaps the most important function defined in this book, and is shown in Figure 6.1.

Some things to note about *perform*:

1. The *Context* is the running "state" of the performance, and gets updated in several different ways. For example, the interpretation of the *Tempo* constructor involves scaling *dt* appropriately and updating the *DurT* field of the context.

2. The interpretation of notes and phrases is player dependent. Ultimately a single note is played by the *playNote* function, which takes the player as an argument. Similarly, phrase interpretation is also player dependent, reflected in the use of *interpPhrase*. Precisely how these two functions work is described in Section 6.2.

[1] We don't need a mapping from *InstrumentNames* to instruments, since this is handled in the translation from a performance into MIDI, which is discussed in Chapter **??**.

```
perform :: PMap a → Context a → Music a → Performance
perform pmap
  c@Context{cTime = t, cPlayer = pl, cDur = dt, cKey = k} m =
  case m of
     Primitive (Note d p) → playNote pl c d p
     Primitive (Rest d) → [ ]
     m1 :+: m2 → perform pmap c m1 ++
                    perform pmap (c{cTime = t + dur m1 * dt}) m2
     m1 :=: m2 → merge (perform pmap c m1) (perform pmap c m2)
     Modify (Tempo r) m → perform pmap (c{cDur = dt / r}) m
     Modify (Transpose p) m → perform pmap (c{cKey = k + p}) m
     Modify (Instrument i) m → perform pmap (c{cInst = i}) m
     Modify (Player pn) m → perform pmap (c{cPlayer = pmap pn}) m
     Modify (Phrase pa) m → interpPhrase pl pmap c pa m
```

Figure 6.1: An abstract *perform* function

3. The *DurT* component of the context is the duration, in seconds, of one whole note. To make it easier to compute, we can define a "metronome" function that, given a standard metronome marking (in beats per minute) and the note type associated with one beat (quarter note, eighth note, etc.) generates the duration of one whole note:

   ```
   metro :: Int → Dur → DurT
   metro setting dur = 60 / (fromIntegral setting * dur)
   ```

   Thus, for example, *metro* 96 *qn* creates a tempo of 96 quarter notes per minute.

4. In the treatment of (:+:), note that the sub-sequences are appended together, with the start time of the second argument delayed by the duration of the first. The function *dur* (defined in Section 5.5) is used to compute this duration. However, this results in a quadratic time complexity for *perform*. A more efficient solution is to have *perform* compute the duration directly, returning it as part of its result. This version of *perform* is shown in Figure 6.2.

5. The sub-sequences derived from the arguments to (:=:) are merged into a time-ordered stream. The definition of *merge* is given below.

```
merge :: Performance → Performance → Performance

merge a@(e1 : es1) b@(e2 : es2) =
      if e1 < e2 then e1 : merge es1 b
                 else e2 : merge a es2
merge [ ] es2 = es2
merge es1 [ ] = es1
```

Note that *merge* compares entire events rather than just start times. This is to ensure that it is commutative, a desirable condition for some of the proofs used later in the text. Here is a more efficient version that will work just as well in practice:

```
merge a@(e1 : es1) b@(e2 : es2) =
  if eTime e1 < eTime e2 then e1 : merge es1 b
                         else e2 : merge a es2
merge [ ] es2 = es2
merge es1 [ ] = es1
```

6.2 Players

Recall from Section 2.2 the definition of the *Control* data type:

```
data Control =
        Tempo Rational          -- scale the tempo
      | Transpose AbsPitch      -- transposition
      | Instrument InstrumentName     -- intrument label
      | Phrase [PhraseAttribute]      -- phrase attributes
      | Player PlayerName       -- player label
    deriving (Show, Eq, Ord)

type PlayerName = String
```

We mentioned, but did not define, the *PhraseAttribute* data type, shown now fully in Figure 6.3. These attributes give us great flexibility in the interpretation process, because they can be interpreted by different players in different ways. For example, how should "legato" be interpreted in a performance? Or "diminuendo?" Different players interpret things in different ways, of course, but even more fundamental is the fact that a pianist, for example, realizes legato in a way fundamentally different from the way a

```
perform :: PMap a → Context a → Music a → Performance
perform pmap c m = fst (perf pmap c m)

perf :: PMap a → Context a → Music a → (Performance, DurT)
perf pmap
  c@Context{cTime = t, cPlayer = pl, cDur = dt, cKey = k}m =
  case m of
    Primitive (Note d p) → (playNote pl c d p, d ∗ dt)
    Primitive (Rest d) → ([ ], d ∗ dt)
    m1 :+: m2 → let (pf1, d1) = perf pmap c m1
                    (pf2, d2) = perf pmap (c{cTime = t + d1}) m2
                in (pf1 ⧺ pf2, d1 + d2)
    m1 :=: m2 → let (pf1, d1) = perf pmap c m1
                    (pf2, d2) = perf pmap c m2
                in (merge pf1 pf2, max d1 d2)
    Modify (Tempo r) m → perf pmap (c{cDur = dt / r}) m
    Modify (Transpose p) m → perf pmap (c{cKey = k + p}) m
    Modify (Instrument i) m → perf pmap (c{cInst = i}) m
    Modify (Player pn) m → perf pmap (c{cPlayer = pmap pn}) m
    Modify (Phrase pas) m → interpPhrase pl pmap c pas m
```

Figure 6.2: The “real” *perform* function

```
data PhraseAttribute = Dyn Dynamic
                     | Tmp Tempo
                     | Art Articulation
                     | Orn Ornament
     deriving (Eq, Ord, Show)

data Dynamic = Accent Rational | Crescendo Rational | Diminuendo Rational
             | StdLoudness StdLoudness | Loudness Rational
     deriving (Eq, Ord, Show)

data StdLoudness = PPP | PP | P | MP | SF | MF | NF | FF | FFF
     deriving (Eq, Ord, Show, Enum)

data Tempo = Ritardando Rational | Accelerando Rational
     deriving (Eq, Ord, Show)

data Articulation = Staccato Rational | Legato Rational | Slurred Rational
                  | Tenuto | Marcato | Pedal | Fermata | FermataDown | Breath
                  | DownBow | UpBow | Harmonic | Pizzicato | LeftPizz
                  | BartokPizz | Swell | Wedge | Thumb | Stopped
     deriving (Eq, Ord, Show)

data Ornament = Trill | Mordent | InvMordent | DoubleMordent
              | Turn | TrilledTurn | ShortTrill
              | Arpeggio | ArpeggioUp | ArpeggioDown
              | Instruction String | Head NoteHead
     deriving (Eq, Ord, Show)

data NoteHead = DiamondHead | SquareHead | XHead | TriangleHead
              | TremoloHead | SlashHead | ArtHarmonic | NoHead
     deriving (Eq, Ord, Show)
```

Figure 6.3: Phrase Attributes

violinist does, because of differences in their instruments. Similarly, diminuendo on a piano and diminuendo on a harpsichord are different concepts.

With a slight stretch of the imagination, we can even consider a "notator" of a score as a kind of player: exactly how the music is rendered on the written page may be a personal, stylized process. For example, how many, and which staves should be used to notate a particular instrument?

In any case, to handle these issues, Haskore has a notion of a *player* that "knows" about differences with respect to performance and notation. A Haskore player is a 4-tuple consisting of a name and three functions: one for interpreting notes, one for phrases, and one for producing a properly notated score.

```
data Player a = MkPlayer{pName :: PlayerName,
                         playNote :: NoteFun a,
                         interpPhrase :: PhraseFun a,
                         notatePlayer :: NotateFun a}
     deriving Show

type NoteFun a = Context a → Dur → a → Performance
type PhraseFun a = PMap a → Context a → [PhraseAttribute]
                   → Music a → (Performance, DurT)
type NotateFun a = ()
```

The last line above is because notation is currently not implemented.

6.2.1 Examples of Player Construction

In order to provide the most flexibility, we define attributes for individual notes:

```
data NoteAttribute = Volume Integer      -- by MIDI convention: 0=min, 127=max
                   | Fingering Integer
                   | Dynamics String
                   | PFields [Float]
     deriving (Eq, Show)
```

Our goal then is to define a player for music values of type:

```
type Note1 = (Pitch, [NoteAttribute])
type Music1 = Music Note1
```

A "default player" called *defPlayer* (not to be confused with a "deaf player"!) is defined for use when none other is specified in the score; it also functions

```
defPlayer :: Player (Pitch, [NoteAttribute])
defPlayer = MkPlayer{ pName = "Default",
                      playNote = defPlayNote defNasHandler,
                      interpPhrase = defInterpPhrase defPasHandler,
                      notatePlayer = defNotatePlayer () }

defPlayNote :: (Context (Pitch, [a]) → a → Event → Event)
             → NoteFun (Pitch, [a])
defPlayNote nasHandler
  c@(Context cTime cPlayer cInst cDur cKey cVol) d (p, nas) =
      [foldr (nasHandler c)
             (Event{ eTime = cTime, eInst = cInst,
                     ePitch = absPitch p + cKey,
                     eDur = d ∗ cDur, eVol = cVol,
                     pFields = [] })
             nas]

defNasHandler :: Context a → NoteAttribute → Event → Event
defNasHandler c (Volume v) ev = ev{ eVol = v }
defNasHandler c (PFields pfs) ev = ev{ pFields = pfs }
defNasHandler _ _ ev = ev

defInterpPhrase :: (PhraseAttribute → Performance → Performance)
                 → PhraseFun a
defInterpPhrase pasHandler pmap context pas m =
      let (pf, dur) = perf pmap context m
      in (foldr pasHandler pf pas, dur)

defPasHandler :: PhraseAttribute → Performance → Performance
defPasHandler (Dyn (Accent x)) =
               map (λe → e{ eVol = round (x ∗ fromIntegral (eVol e)) })
defPasHandler (Art (Staccato x)) = map (λe → e{ eDur = x ∗ eDur e })
defPasHandler (Art (Legato x)) = map (λe → e{ eDur = x ∗ eDur e })
defPasHandler _ = id

defNotatePlayer :: a → ()
defNotatePlayer _ = ()
```

Figure 6.4: Definition of default Player *defPlayer*.

as a base from which other players can be derived. *defPlayer* responds only to the *Volume* note attribute and to the *Accent*, *Staccato*, and *Legato* phrase attributes. It is defined in Figure 6.4. Before reading this code, recall how players are invoked by the *perform* function defined in the last section; in particular, note the calls to *playNote* and *interpPhrase*. Then note:

1. *defPlayNote* is the only function (even in the definition of *perform*) that actually generates an event. It also modifies that event based on an interpretation of each note attribute by the function *defHasHandler*.

2. *defNasHandler* only recognizes the *Volume* attribute, which it uses to set the event volume accordingly.

3. *defInterpPhrase* calls (mutually recursively) *perform* to interpret a phrase, and then modifies the result based on an interpretation of each phrase attribute by the function *defPasHandler*.

4. *defPasHandler* only recognizes the *Accent*, *Staccato*, and *Legato* phrase attributes. For each of these it uses the numeric argument as a "scaling" factor of the volume (for *Accent*) and duration (for *Staccato* and *Lagato*). Thus *Modify* (*Phrase* [*Legato* (5 / 4)]) *m* effectively increases the duration of each note in *m* by 25% (without changing the tempo).

It should be clear that much of the code in Figure 6.4 can be re-used in defining a new player. For example, to define a player *weird* that interprets note attributes just like *defPlayer* but behaves differently with respect to phrase attributes, we could write:

```
weird :: Player (Pitch, [NoteAttribute])
weird = MkPlayer{ pName = "newPlayer",
                  playNote = defPlayNote defNasHandler,
                  interpPhrase = defInterpPhrase myPasHandler,
                  notatePlayer = defNotatePlayer () }
```

and then supply a suitable definition of *myPasHandler*. That definition could also re-use code, in the following sense: suppose we wish to add an interpretation for *Crescendo*, but otherwise have *myPasHandler* behave just like *defPasHandler*.

```
myPasHandler :: PhraseAttribute → Performance → Performance
myPasHandler (Dyn (Crescendo x)) pf = ...
myPasHandler pa pf = defPasHandler pa pf
```

Exercise 6.1 Fill in the ... in the definition of *myPasHandler* according to the following strategy: Gradually scale the volume of each event in the performance by a factor of 1 through $1 + x$, using linear interpolation.

Exercise 6.2 Choose some of the other phrase attributes and provide interpretations of them, such as *Diminuendo*, *Slurred*, *Trill*, etc. (The *trill* functions from section 5.8 may be useful here.)

Figure 6.5 defines a relatively sophisticated player called *fancyPlayer* that knows all that *defPlayer* knows, and much more. Note that *Slurred* is different from *Legato* in that it doesn't extend the duration of the *last* note(s). The behavior of *Ritardando* x can be explained as follows. We'd like to "stretch" the time of each event by a factor from 0 to x, linearly interpolated based on how far along the musical phrase the event occurs. I.e., given a start time t_0 for the first event in the phrase, total phrase duration D, and event time t, the new event time t' is given by:

$$t' = (1 + \frac{t - t_0}{D} x)(t - t_0) + t_0$$

Further, if d is the duration of the event, then the end of the event $t + d$ gets stretched to a new time t'_d given by:

$$t'_d = (1 + \frac{t + d - t_0}{D} x)(t + d - t_0) + t_0$$

The difference $t'_d - t'$ gives us the new, stretched duration d', which after simplification is:

$$d' = (1 + \frac{2(t - t_0) + d}{D} x)d$$

Accelerando behaves in exactly the same way, except that it shortens event times rather than lengthening them. And, a similar but simpler strategy explains the behaviors of *Crescendo* and *Diminuendo*.

```
fancyPlayer :: Player (Pitch, [NoteAttribute])
fancyPlayer = MkPlayer{ pName = "Fancy",
                        playNote = defPlayNote defNasHandler,
                        interpPhrase = fancyInterpPhrase,
                        notatePlayer = defNotatePlayer ()}
fancyInterpPhrase :: PhraseFun a
fancyInterpPhrase pmap c [] m = perf pmap c m
fancyInterpPhrase pmap c@Context{ cTime = t, cPlayer = pl, cInst = i,
                                  cDur = dt, cKey = k, cVol = v}
                  (pa : pas) m =
 let pfd@(pf, dur) = fancyInterpPhrase pmap c pas m
     loud x = fancyInterpPhrase pmap c (Dyn (Loudness x) : pas) m
     stretch x = let t0 = eTime (head pf); r = x / dur
                     upd (e@Event{ eTime = t, eDur = d}) =
                       let dt = t − t0
                           t' = (1 + dt ∗ r) ∗ dt + t0
                           d' = (1 + (2 ∗ dt + d) ∗ r) ∗ d
                       in e{ eTime = t', eDur = d'}
                 in (map upd pf, (1 + x) ∗ dur)
     inflate x = let t0 = eTime (head pf); r = x / dur
                     upd (e@Event{ eTime = t, eVol = v}) =
                       e{ eVol = (round ∘ fromRational) (1 + (t − t0) ∗ r) ∗ v}
                 in (map upd pf, dur)
 in case pa of
    Dyn (Accent x) → (map (λe → e{ eVol = round (x ∗ fromIntegral (eVol e))}) pf, dur)
    Dyn (StdLoudness l) →
         case l of
           PPP → loud 40; PP → loud 50; P → loud 60
           MP → loud 70; SF → loud 80; MF → loud 90
           NF → loud 100; FF → loud 110; FFF → loud 120
    Dyn (Loudness x) → fancyInterpPhrase pmap c{ cVol = (round ∘ fromRational) x}pas m
    Dyn (Crescendo x) → inflate x; Dyn (Diminuendo x) → inflate (−x)
    Tmp (Ritardando x) → stretch x; Tmp (Accelerando x) → stretch (−x)
    Art (Staccato x) → (map (λe → e{ eDur = x ∗ eDur e}) pf, dur)
    Art (Legato x) → (map (λe → e{ eDur = x ∗ eDur e}) pf, dur)
    Art (Slurred x) →
        let lastStartTime = foldr (λe t → max (eTime e) t) 0 pf
            setDur e = if eTime e < lastStartTime
                       then e{ eDur = x ∗ eDur e}
                       else e
        in (map setDur pf, dur)
    Art _ → pfd
    Orn _ → pfd
        -- Design Bug: To do these right we need to keep the KEY SIGNATURE
        -- around so that we can determine, for example, what the trill note is.
        -- Alternatively, provide an argument to Trill to carry this info.
```

Figure 6.5: Definition of Player *fancyPlayer*.

Chapter 7

Self-Similar Music

```
module Haskore.SelfSimilar where
import Haskore
```

In this chapter we will explore the notion of *self-similar* music—i.e. musical structures that have patterns that repeat themselves recursively in interesting ways. There are many approaches to generating self-similar structures, the most general being *fractals*, which have been used to generate not just music, but also graphical images. We will delay a general treatment of fractals, however, and will instead focus on more specialized notions of self-similarity, notions that we conceive of musically, and then manifest as Haskell programs.

7.1 Self-Similar Melody

Here is the first notion of self-similar music that we will consider: Begin with a very simple melody of n notes. Now duplicate this melody n times, playing each in succession, but first perform the following transformations: transpose the ith melody by an amount proportional to the pitch of the ith note in the original melody, and scale its tempo by a factor proportional to the duration of the ith note. For example, Figure 7.1 shows the result of applying this process once to a four-note melody. Now imagine that this process is repeated infinitely often. For a melody whose notes are all shorter than a whole note, it yields an infinitely dense melody of infinitesimally shorter notes. To make the result playable, however, we will stop the process at some pre-determined level.

How can this be represented in Haskell? A *tree* seems like it would be a logical choice; let's call it a *Cluster*:

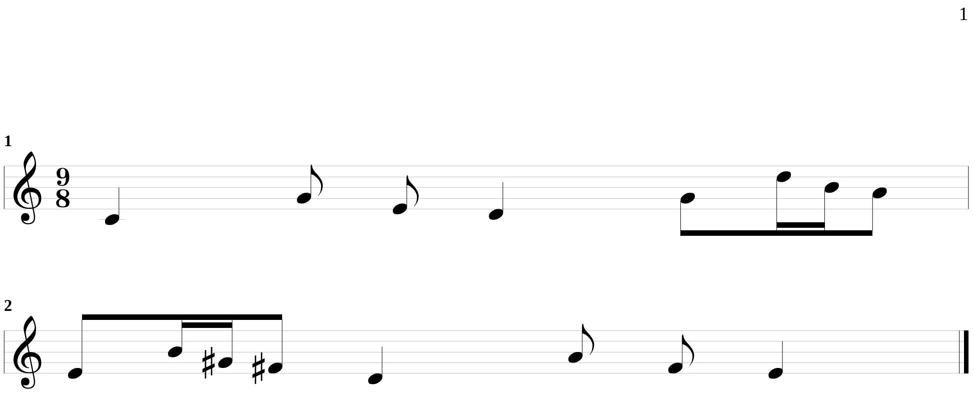


Figure 7.1: An Example of Self-Similar Music

```
data Cluster = Cluster SNote [Cluster]
type SNote = (Dur, AbsPitch)
```

This particular kind of tree happens to be called a *rose tree* []. An *SNote* is just a "simple note," a duration paired with an absolute pitch. We prefer to stick with absolute pitches in creating the self-similar structure, and will convert the result into a normal *Music* value only after we are done.

The sequence of *SNote*s at each level of the cluster is the melodic fragment for that level. The very top cluster will contain a "dummy" note, whereas the next level will contain the original melody, the next level will contain one iteration of the process described above (e.g. the melody in Figure 7.1), and so forth.

To achieve this we will define a function *selfSim* that takes the initial melody as argument and generates an infinitely deep cluster:

```
selfSim :: [SNote] → Cluster
selfSim pat = Cluster (0, 0) (map mkCluster pat)
    where mkCluster note
              = Cluster note (map (mkCluster ∘ addMult note) pat)

addMult :: SNote → SNote → SNote
addMult (d0, p0) (d1, p1) = (d0 ∗ d1, p0 + p1)
```

Note that *selfSim* itself is not recursive, but *mkCluster* is.

Next, we define a function to skim off the notes at the *n*th level, or *n*th "fringe," of a cluster:

```
fringe :: Int → Cluster → [SNote]
fringe 0 (Cluster note cls) = [note]
fringe n (Cluster note cls) = concatMap (fringe (n − 1)) cls
```

Details: *concatMap* is defined in the Standard Prelude as:

```
concatMap :: (a → [b]) → [a] → [b]
concatMap f = concat ∘ map f
```

Also recall that *concat* appends together a list of lists, and is defined in the Prelude as:

```
concat :: [[a]] → [a]
concat = foldr (++) []
```

All that is left to do is convert this into a *Music* value that we can play:

```
simToMusic :: [SNote] → Music Pitch
simToMusic ss = let mkNote (d, ap) = note d (pitch ap)
                in line (map mkNote ss)
```

We can define this with a bit more elegance as follows:

```
simToMusic :: [SNote] → Music Pitch
simToMusic = line ∘ map mkNote

mkNote :: (Dur, AbsPitch) → Music Pitch
mkNote (d, ap) = note d (pitch ap)
```

The increased modularity will allow us to reuse *mkNote* later in the chapter.

Putting it all together, we can define a function that takes an initial pattern, a level, a number of pitches to transpose the result, and a tempo scaling factor, to yield a final result:

```
ss pat n tr te = transpose tr (tempo te (simToMusic (fringe n (selfSim pat))))
```

Here are some example compositions:

```
p1 :: [SNote]
p1 = [(hn, 3), (qn, 4), (qn, 0), (wn, 6)]

ss1 = ss p1 4 50 (3 / 2)

ss1a = let l1 = instrument Flute ss1
           l2 = instrument AcousticBass (transpose (−12) (revM ss1))
```

```
          in l1 :=: l2

        -- Note that the flute and bass lines are the reverse of one another.

    p2 = [(dqn, 0), (qn, 4)]
    p3 = [(6 / 10, 2), (13 / 10, 5), (wn, 0), (9 / 10, 7)]
    p4 = [(hn, 3), (hn, 8), (hn, 22), (qn, 4), (qn, 7), (qn, 21),
          (qn, 0), (qn, 5), (qn, 15), (wn, 6), (wn, 9), (wn, 19)]

    ss2 = ss p2 6 50 (1 / 30)
    ss3 = ss p3 4 50 20
    ss4 = ss p4 3 50 8
```

Exercise 7.1 Experiment with this idea futher, using other melodic seeds, exploring different depths of the clusters, and so on.

Exercise 7.2 Note that *concat* is defined as *foldr* (++) [ ], which means that it takes a number of steps proportional to the sum of the lengths of the lists being concatenated; we cannot do any better than this. (If *foldl* were used instead, the number of steps would be proportional to the number of lists times their average length.)

However, *fringe* is not very efficient, for the following reason: *concat* is being used over and over again, like this:

```
    concat [concat [...], concat [...], concat [...]]
```

This causes a number of steps proportional to the depth of the tree times the length of the sub-lists; clearly not optimal.

Define a version of *fringe* that is linear in the total length of the final list.

7.2 Self-Similar Harmony

In the last section we used a melody as a seed, and created longer melodies from it. Another idea is to stack the melodies vertically. Specifically, suppose we redefine *fringe* in such a way that it does not concatenate the sub-clusters together:

```
    fringe′ :: Int → Cluster → [[SNote]]
    fringe′ 0 (Cluster note cls) = [[note]]
    fringe′ n (Cluster note cls) = map (fringe (n − 1)) cls
```

Note that this strategy is only applied to the top level—below that we use fringe. Thus the type of the result is [[*SNote*]], i.e. a list of lists of notes.

We can convert the individual lists into melodies, and play the melodies all together, like this:

```
simToMusic' :: [[SNote]] → Music Pitch
simToMusic' = chord ∘ map (line ∘ map mkNote)
```

Finally, we can define a function akin to *ss* defined earlier:

```
ss' pat n tr te = transpose tr (tempo te (simToMusic' (fringe' n (selfSim pat))))
```

Using the same patterns as used earlier, here are some sample compositions:

```
ss1' = ss' p1 4 50 (3 / 2)
ss2' = ss' p2 4 50 4
ss3' = ss' p3 4 50 20
ss4' = ss' p4 3 50 8
```

And a new one, based on a major triad:

```
ss5 = ss p5 4 45 (1 / 500)
ss5' = ss' p5 4 45 (1 / 500)
p5 = [(en, 4), (sn, 7), (en, 0)]
```

Note the need to scale the tempo back drastically, due to the short durations of the starting notes.

7.3 Other Self-Similar Structures

The reader will observe that our notion of "self-similar harmony" did *not* involve changing the structure of the *Cluster* data type, nor the algorithm for computing the sub-structures (as captured in *selfSim*). All that we did was interpret the result differently. This is a common characteristic of algorithmic music compisition—the same mathematical or computational structure is interpreted in different ways to yield musically different results.

For example, instead of the above strategy for playing melodies in parallel, we could play entire levels of the *Cluster* in parallel, where the number of levels that we choose is given as a parameter. We leave this idea and others as exercises for the reader.

Exercise 7.3 Devise a version of *simToMusic* that constructs a *Music* value as outlined above. Specifically, given a parameter *n*, *simToMusic n pat* plays the first *n* levels of the cluster generated by *pat* in parallel.

Exercise 7.4 Devise some other variant of self-similar music, and encode it in Haskell. In particular, consider structures that are different from those generated by the *selfSim* function.

Chapter 8

Proof by Induction

In this chapter we will study a powerful proof technique based on *mathematical induction.* With it we will be able to prove complex and important properties of programs that cannot be accomplished with proof-by-calculation alone. The inductive proof method is one of the most powerful and common methods for proving program properties.

8.1 Induction and Recursion

Induction is very closely related to *recursion.* In fact, in certain contexts the terms are used interchangeably; in others, one is preferred over the other primarily for historical reasons. Think of them as being duals of one another: induction is used to describe the process of starting with something small and simple, and building up from there, whereas recursion describes the process of starting with something large and complex, and working backward to the simplest case.

For example, although we have previously used the phrase *recursive data type*, in fact data types are often described *inductively*, such as a list:

> A *list* is either empty, or it is a pair consisting of a value and another list.

On the other hand, we usually describe functions that manipulate lists, such as *map* and *foldr*, as being recursive. This is because when you apply a function such as *map*, you apply it initially to the whole list, and work backwards toward [ ].

But these differences between induction and recursion run no deeper: they are really just two sides of the same coin.

This chapter is about *inductive properties* of programs (but based on the above argument could just as rightly be called *recursive properties*) that are not usually proven via calculation alone. Proving inductive properties usually involves the inductive nature of data types and the recursive nature of functions defined on the data types.

As an example, suppose that p is an inductive property of a list. In other words, $p\ (l)$ for some list l is either true or false (no middle ground!). To prove this property inductively, we do so based on the length of the list: starting with length 0, we first prove $p\ ([\,])$ (using our standard method of proof-by-calculation).

Now for the key step: assume for the moment that $p\ (xs)$ is true for any list xs whose length is less than or equal to n. Then if we can prove (via calculation) that $p\ (x : xs)$ is true for any x—i.e. that p is true for lists of length $n+1$—then the claim is that p is true for lists of *any* (finite) length.

Why is this so? Well, from the first step above we know that p is true for length 0, so the second step tells us that it's also true for length 1. But if it's true for length 1 then it must also be true for length 2; similarly for lengths 3, 4, etc. So p is true for lists of any length!

(It it important to realize, however, that a property being true for every finite list does not necessarily imply that it is true for every infinite list. The property "the list is finite" is a perfect example of this! We will see how to prove properties of infinite lists in Chapter **??**.)

To summarize, to prove a property p by induction on the length of a list, we proceed in two steps:

1. Prove $p\ ([\,])$ (this is called the *base step*).
2. Assume that $p\ (xs)$ is true (this is called the *induction hypothesis*, and prove that $p\ (x : xs)$ is true (this is called the *induction step*).

8.2 Examples of List Induction

Ok, enough talk, let's see this idea in action. Recall in Section 3.1 the following property about *foldr*:

$$(\forall xs)\ \ foldr\ (:)\ [\,]\ xs \Longrightarrow xs$$

We will prove this by induction on the length of xs. Following the ideas above, we begin with the base step by proving the property for length 0; i.e. for $xs = [\,]$:

$$\mathit{foldr}\ (:)\ [\,]\ [\,] \Rightarrow [\,]$$

This step is immediate from the definition of *foldr*. Now for the induction step: we first *assume* that the property is true for all lists xs of length n, and then prove the property for list $x : xs$. Again proceeding by calculation:

$$\begin{array}{l} \mathit{foldr}\ (:)\ [\,]\ (x : xs) \\ \Rightarrow x : \mathit{foldr}\ (:)\ [\,]\ xs \\ \Rightarrow x : xs \end{array}$$

And we are done; the induction hypothesis is what justifies the second step.

Now let's do something a bit harder. Suppose we are interested in proving the following property:

$$(\forall xs, ys)\ \ \mathit{length}\ (xs +\!\!+\ ys) = \mathit{length}\ xs + \mathit{length}\ ys$$

Our first problem is to decide which list to perform the induction over. A little thought (in particular, a look at how the definitions of *length* and $(+\!\!+)$ are structured) should convince you that xs is the right choice. (If you do not see this, you are encouraged to try the proof by induction over the length of ys!) Again following the ideas above, we begin with the base step by proving the property for length 0; i.e. for $xs = [\,]$:

$$\begin{array}{l} \mathit{length}\ ([\,] +\!\!+\ ys) \\ \Rightarrow \mathit{length}\ ys \\ \Rightarrow 0 + \mathit{length}\ ys \\ \Rightarrow \mathit{length}\ [\,] + \mathit{length}\ ys \end{array}$$

For the induction step, we first assume that the property is true for all lists xs of length n, and then prove the property for list $x : xs$. Again proceeding by calculation:

$$\begin{array}{l} \mathit{length}\ ((x : xs) +\!\!+\ ys) \\ \Rightarrow \mathit{length}\ (x : (xs +\!\!+\ ys)) \\ \Rightarrow 1 + \mathit{length}\ (xs +\!\!+\ ys) \\ \Rightarrow 1 + (\mathit{length}\ xs + \mathit{length}\ ys) \\ \Rightarrow (1 + \mathit{length}\ xs) + \mathit{length}\ ys \\ \Rightarrow \mathit{length}\ (x : xs) + \mathit{length}\ ys \end{array}$$

And we are done. The transition from the 3rd line to the 4th is where we used the induction hypothesis.

8.3 Proving Function Equivalences

At this point it is a simple matter to return to Chapter 3 and supply the proofs that functions defined using *map* and *fold* are equivalent to the recursive versions. In particular, let's prove first that:

$$toAbsPitches\ ps = map\ absPitch\ ps$$

for any finite list *ps*, where:

$$toAbsPitches\ [\,] = [\,]$$
$$toAbsPitches\ (p : ps) = absPitch\ p : toAbsPitches\ ps$$

We proceed by induction, starting with the base case $ps = [\,]$:

$$toAbsPitches\ [\,]$$
$$\Rightarrow [\,]$$
$$\Rightarrow map\ absPitch\ [\,]$$

Next we assume that $toAbsPitches\ ps = map\ absPitch\ ps$ holds, and try to prove that $toAbsPitches\ (p : ps) = map\ absPitch\ (p : ps)$ (note the use of the induction hypothesis in the second step):

$$toAbsPitches\ (p : ps)$$
$$\Rightarrow absPitch\ p : toAbsPitches\ ps$$
$$\Rightarrow absPitch\ p : map\ absPitch\ ps$$
$$\Rightarrow map\ absPitch\ (p : ps)$$

The proof that $toPitches\ aps = map\ pitch\ aps$ is very similar, and is left as an exercise.

For a proof involving *foldr*, recall from Section 3.4 this recursive definition of *line*:

$$line\ [\,] = rest\ 0$$
$$line\ (m : ms) = m \mathbin{:+:} line\ ms$$

and this non-recursive version:

$$line = foldr\ (:+:)\ (rest\ 0)$$

We can prove that these definitions are equivalent by induction. First the base case:

$$line\ [\,]$$
$$\Rightarrow rest\ 0$$
$$\Rightarrow foldr\ (:+:)\ (rest\ 0)\ [\,]$$

Then the induction step:

$\mathit{line}\ (m : \mathit{ms})$
$\Rightarrow m :\!+\!: \mathit{line}\ \mathit{ms}$
$\Rightarrow m :\!+\!: \mathit{foldr}\ (:\!+\!:)\ (\mathit{rest}\ 0)\ \mathit{ms}$
$\Rightarrow \mathit{foldr}\ (:\!+\!:)\ (\mathit{rest}\ 0)\ (m : \mathit{ms})$

The proofs of equivalence of the definitions of *chord*, *maxPitch*, and *listSum* from Section 3.4 are similar, and left as exercises.

These proofs were in fact quite easy. For something more challenging, consider the definition of *reverse* given in Section 3.5:

$\mathit{reverse1}\ [\,] = [\,]$
$\mathit{reverse1}\ (x : \mathit{xs}) = \mathit{reverse1}\ \mathit{xs} +\!\!+ [x]$

and the version given in Section 4.1:

$\mathit{reverse2}\ \mathit{xs} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ \mathit{xs}$

We would like to show that these are the same; i.e. that $\mathit{reverse1}\ \mathit{xs} = \mathit{reverse2}\ \mathit{xs}$ for any finite list *xs*. In carrying out this proof two new ideas will be demonstrated, the first being that induction can be used to prove the equivalence of two programs. The second is the need for an *auxiliary property* which is proved independently of the main result.

The base case is easy, as it often is:

$\mathit{reverse1}\ [\,]$
$\Rightarrow [\,]$
$\Rightarrow \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ [\,]$
$\Rightarrow \mathit{reverse2}\ [\,]$

Assume now that $\mathit{reverse1}\ \mathit{xs} = \mathit{reverse2}\ \mathit{xs}$. The induction step proceeds as follows:

$\mathit{reverse1}\ (x : \mathit{xs})$
$\Rightarrow \mathit{reverse1}\ \mathit{xs} +\!\!+ [x]$
$\Rightarrow \mathit{reverse2}\ \mathit{xs} +\!\!+ [x]$
$\Rightarrow \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ \mathit{xs} +\!\!+ [x]$
$\Rightarrow ???$

But now what do we do? Intuitively, it seems that the following property, which we will call property (1), should hold:

$\mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ \mathit{xs} +\!\!+ [x]$
$\Rightarrow \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ (x : \mathit{xs})$

in which case we could complete the proof as follows:

...
$\Rightarrow$ *foldl* (*flip* (:)) [ ] *xs* ++ [*x*]
$\Rightarrow$ *foldl* (*flip* (:)) [ ] (*x* : *xs*)
$\Rightarrow$ *reverse2* (*x* : *xs*)

The ability to see that if we could just prove one thing, then perhaps we could prove another, is a useful skill in conducting proofs. In this case we have reduced the overall problem to one of proving property (1), which simplifies the structure of the proof, although not necessarily the difficulty. These auxiliary properties are often called *lemmas* in mathematics, and in many cases their proofs become the most important contributions, since they are often at the heart of a problem.

In fact if you try to prove property (1) directly, you will run into a problem, namely that it is not *general* enough. So first let's generalize property (1) (while renaming *x* to *y*), as follows:

foldl (*flip* (:)) *ys xs* ++ [*y*]
$\Rightarrow$ *foldl* (*flip* (:)) (*ys* ++ [*y*]) *xs*

Let's call this property (2). If (2) is true for any finite *xs* and *ys*, then property (1) is also true, because:

foldl (*flip* (:)) [ ] *xs* ++ [*x*]
$\Rightarrow$ {*property* (2)}
foldl (*flip* (:)) ([ ] ++ [*x*]) *xs*
$\Rightarrow$ {*unfold* (++)}
foldl (*flip* (:)) [*x*] *xs*
$\Rightarrow$ {*fold* (*flip* (:))}
foldl (*flip* (:)) (*flip* (:) [ ] *x*) *xs*
$\Rightarrow$ {*fold foldl*}
foldl (*flip* (:)) [ ] (*x* : *xs*)

You are encouraged to try proving property (1) directly, in which case you will likely come to the same conclusion, namely that the property needs to be generalized. This is not always easy to see, but is sometimes an important step is constructing a proof, because, despite being somewhat counterintuitive, it is often the case that making a property more general (and therefore more powerful) makes it easier to prove.

In any case, how do we prove property (2)? Using induction, of course! Setting *xs* to [ ], the base case is easy:

$$
\begin{array}{l}
\mathit{foldl}\ (\mathit{flip}\ (:))\ \mathit{ys}\ [\,]\ +\!\!+\ [y] \\
\Rightarrow \{\mathit{unfold\ foldl}\} \\
\mathit{ys}\ +\!\!+\ [y] \\
\Rightarrow \{\mathit{fold\ foldl}\} \\
\mathit{foldl}\ (\mathit{flip}\ (:))\ (\mathit{ys}\ +\!\!+\ [y])\ [\,]
\end{array}
$$

and the induction step proceeds as follows:

$$
\begin{array}{l}
\mathit{foldl}\ (\mathit{flip}\ (:))\ \mathit{ys}\ (x : \mathit{xs})\ +\!\!+\ [y] \\
\Rightarrow \{\mathit{unfold\ foldl}\} \\
\mathit{foldl}\ (\mathit{flip}\ (:))\ (\mathit{flip}\ (:)\ \mathit{ys}\ x)\ \mathit{xs}\ +\!\!+\ [y] \\
\Rightarrow \{\mathit{unfold\ flip}\} \\
\mathit{foldl}\ (\mathit{flip}\ (:))\ (x : \mathit{ys})\ \mathit{xs}\ +\!\!+\ [y] \\
\Rightarrow \{\mathit{induction\ hypothesis}\} \\
\mathit{foldl}\ (\mathit{flip}\ (:))\ ((x : \mathit{ys})\ +\!\!+\ [y])\ \mathit{xs} \\
\Rightarrow \{\mathit{unfold}\ (+\!\!+)\} \\
\mathit{foldl}\ (\mathit{flip}\ (:))\ (x : (\mathit{ys}\ +\!\!+\ [y]))\ \mathit{xs} \\
\Rightarrow \{\mathit{fold\ foldl}\} \\
\mathit{foldl}\ (\mathit{flip}\ (:))\ (\mathit{ys}\ +\!\!+\ [y])\ (x : \mathit{xs})
\end{array}
$$

8.4 Useful Properties on Lists

There are many useful properties of functions on lists that require inductive proofs. Tables 8.1 and 8.2 list a number of them involving functions used in this text, but their proofs are left as exercises (except for one; see below). You may assume that these properties are true, and use them freely in proving other properties of your programs. In fact, some of these properties can be used to simplify the proof that *reverse1* and *reverse2* are the same; see if you can find them![1]

(Note, by the way, that in the first rule for *map* in Figure 8.1, the type of $\lambda x \rightarrow x$ on the left-hand side is $a \rightarrow b$, whereas on the right-hand side it is $[\,a\,] \rightarrow [\,b\,]$; i.e. these are really two different functions.)

8.4.1 Function Strictness

Note that the last rule for *map* in Figure 8.1 is only valid for *strict* functions. A function f is said to be strict if $f\ \bot = \bot$. Recall from Section 1.2 that $\bot$ is the value associated with a non-terminating computation. So another

[1] More thorough discussions of these properties and their proofs may be found in [BW88, Bir98].

Properties of *map*:

$$\begin{aligned}
&\mathit{map}\ (\lambda x \rightarrow x) = \lambda x \rightarrow x\\
&\mathit{map}\ (f \circ g) = \mathit{map}\ f \circ \mathit{map}\ g\\
&\mathit{map}\ f \circ \mathit{tail} = \mathit{tail} \circ \mathit{map}\ f\\
&\mathit{map}\ f \circ \mathit{reverse} = \mathit{reverse} \circ \mathit{map}\ f\\
&\mathit{map}\ f \circ \mathit{concat} = \mathit{concat} \circ \mathit{map}\ (\mathit{map}\ f)\\
&\mathit{map}\ f\ (\mathit{xs} +\!\!+\ \mathit{ys}) = \mathit{map}\ f\ \mathit{xs} +\!\!+\ \mathit{map}\ f\ \mathit{ys}
\end{aligned}$$

For all strict f:

$$f \circ \mathit{head} = \mathit{head} \circ \mathit{map}\ f$$

Properties of the *fold* functions:

1. If op is associative, and $e\,\text{`}\mathit{op}\text{`}\,x = x$ and $x\,\text{`}\mathit{op}\text{`}\,e = x$ for all x, then for all finite xs:

   $$\mathit{foldr}\ \mathit{op}\ e\ \mathit{xs} = \mathit{foldl}\ \mathit{op}\ e\ \mathit{xs}$$

2. If the following are true:

   $$\begin{aligned}
   &x\,\text{`}\mathit{op1}\text{`}\,(y\,\text{`}\mathit{op2}\text{`}\,z) = (x\,\text{`}\mathit{op1}\text{`}\,y)\,\text{`}\mathit{op2}\text{`}\,z\\
   &x\,\text{`}\mathit{op1}\text{`}\,e = e\,\text{`}\mathit{op2}\text{`}\,x
   \end{aligned}$$

   then for all finite xs:

   $$\mathit{foldr}\ \mathit{op1}\ e\ \mathit{xs} = \mathit{foldl}\ \mathit{op2}\ e\ \mathit{xs}$$

3. For all finite xs:

   $$\mathit{foldr}\ \mathit{op}\ e\ \mathit{xs} = \mathit{foldl}\ (\mathit{flip}\ \mathit{op})\ e\ (\mathit{reverse}\ \mathit{xs})$$

Table 8.1: Some Useful Properties of *map* and *fold*.

Properties of $(\mathbin{+\!\!+})$:

For all xs, ys, and zs:

$$(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs = xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)$$
$$xs \mathbin{+\!\!+} [\,] = [\,] \mathbin{+\!\!+} xs = xs$$

Properties of *take* and *drop*:

For all finite non-negative m and n, and finite xs:

$$\mathit{take}\ n\ xs \mathbin{+\!\!+} \mathit{drop}\ n\ xs = xs$$
$$\mathit{take}\ m \circ \mathit{take}\ n = \mathit{take}\ (\mathit{min}\ m\ n)$$
$$\mathit{drop}\ m \circ \mathit{drop}\ n = \mathit{drop}\ (m + n)$$
$$\mathit{take}\ m \circ \mathit{drop}\ n = \mathit{drop}\ n \circ \mathit{take}\ (m + n)$$

For all finite non-negative m and n such that $n \geqslant m$:

$$\mathit{drop}\ m \circ \mathit{take}\ n = \mathit{take}\ (n - m) \circ \mathit{drop}\ m$$

Properties of *reverse*:

For all finite xs:

$$\mathit{reverse}\ (\mathit{reverse}\ xs) = xs$$
$$\mathit{head}\ (\mathit{reverse}\ xs) = \mathit{last}\ xs$$
$$\mathit{last}\ (\mathit{reverse}\ xs) = \mathit{head}\ xs$$

Table 8.2: Useful Properties of Other Functions Over Lists

way to think about a strict function is that it is one that, when applied to a non-terminating computation, results in a non-terminating computation. For example, the successor function $(+1)$ is strict, because $(+1)\ \bot = \bot + 1 = \bot$. In other words, if you apply $(+1)$ to a non-terminating computation, you end up with a non-terminating computation.

Not all functions in Haskell are strict, and we have to be careful to say on which argument a function is strict. For example, $(+)$ is strict on both of its arguments, which is why the section $(+1)$ is also strict. On the other hand, the constant function:

$$const\ x\ y = x$$

is strict on its first argument (why?), but not its second, because $const\ x\ \bot = x$, for any x.

Details: Understanding strictness requires a careful understanding of Haskell's pattern-matching rules. For example, consider the definition of $(\wedge)$ from the Standard Prelude:

$$\begin{array}{l}(\wedge) :: Bool \rightarrow Bool \rightarrow Bool \\ True \wedge x = x \\ False \wedge \_ = False\end{array}$$

When choosing a pattern to match, Haskell starts with the top, left-most pattern, and works to the right and downward. So in the above, $(\wedge)$ first evaluates its left argument. If that value is $True$, then the first equation succeeds, and the second argument gets evaluated because that is the value that is returned. But if the first argument is $False$, the second equation succeeds. In particular, *it does not bother to evaluate the second argument at all*, and simply returns $False$ as the answer. This means that $(\wedge)$ is strict in its first argument, but not its second.

A more detailed discussion of pattern matching is found in Appendix D.

Let's now look more closely at the last law for map, which says that for all strict f:

$$f \circ head = head \circ map\ f$$

Let's try to prove this property, starting with the base case, but ignoring for now the strictness constraint on f:

$$\begin{array}{l}f\ (head\ [\,]) \\ \Rightarrow f\ \bot\end{array}$$

head [ ] is an error, which you will recall has value $\bot$. So you can see immediately that the issue of strictness might play a role in the proof, because without knowing anything about f, there is no further calculation to be done here. Similarly, if we start with the right-hand side:

$$\begin{array}{l} head\ (map\ f\ [\,]) \\ \Rightarrow head\ [\,] \\ \Rightarrow \bot \end{array}$$

It should be clear that for the base case to be true, it must be that $f\ \bot = \bot$; i.e., f must be strict. Thus we have essentially "discovered" the constraint on the theorem through the process of trying to prove it! (This is not an uncommon phenomenon.)

The induction step is less problematic:

$$\begin{array}{l} f\ (head\ (x : xs)) \\ \Rightarrow f\ x \\ \Rightarrow head\ (f\ x : map\ f\ xs) \\ \Rightarrow head\ (map\ f\ (x : xs)) \end{array}$$

and we are done.

Exercise 8.1 From Chapter 3, prove that:

- $toPitches = map\ pitch$
- $chord = fold\ (:=:)\ (rest\ 0)$
- $maxPitch = fold\ (!!!)\ 0$
- $listSum\ xs = fold\ (+)\ 0$

Exercise 8.2 Prove as many of the properties in Tables 8.1 and 8.2 as you can.

Exercise 8.3 Which of the following functions are strict (if the function takes more than one argument, specify on which arguments it is strict): *reverse*, *simple*, *map*, *tail*, *area*, ($\wedge$), (*True* $\wedge$), (*False* $\wedge$), and the following function:

$$\begin{array}{l} ifFun :: Bool \rightarrow a \rightarrow a \rightarrow a \\ ifFun\ pred\ cons\ alt = \textbf{if}\ pred\ \textbf{then}\ cons\ \textbf{else}\ alt \end{array}$$

[Replace the following section with properties about musical functions. In particular:

$revM\ (revM\ m) = m$
...

Also prove or leave as exercise the fact that the two version of perform are the same.]

8.5 Induction on Other Data Types

Proof by induction is not limited to lists. For example, we can use it to reason about natural numbers.[2] Suppose we define an exponentiation function as follows:

$(\hat{}) :: Integer \rightarrow Integer \rightarrow Integer$
$x\,\hat{}\,0 = 1$
$x\,\hat{}\,n = x * x\,\hat{}\,(n-1)$

> **Details:** $(*)$ is defined in the Standard Prelude to have precedence level 7, and recall that if no **infix** declaration is given for an operator it defaults to precedence level 9, which means that $(\hat{})$ has precedence level 9, which is higher than that for $(*)$. Therefore no parentheses are needed to disambiguate the last line in the definition above, which corresponds nicely to mathematical convention.

Now suppose that we want to prove that:

$$(\forall x, n \geqslant 0, m \geqslant 0) \quad x\,\hat{}\,(n+m) = x\,\hat{}\,n * x\,\hat{}\,m$$

We proceed by induction on n, beginning with $n = 0$:

$x\,\hat{}\,(0+m)$
$\Rightarrow x\,\hat{}\,m$
$\Rightarrow 1 * (x\,\hat{}\,m)$
$\Rightarrow x\,\hat{}\,0 * x\,\hat{}\,m$

Next we assume that the property is true for numbers less than or equal to n, and prove it for $n + 1$:

[2] Indeed, one could argue that a proof by induction over finite lists is really an induction over natural numbers, since it is an induction over the *length* of the list, which is a natural number.

$$\begin{array}{l} x\,\hat{}\,((n+1)+m) \\ \Rightarrow x * x\,\hat{}\,(n+m) \\ \Rightarrow x * (x\,\hat{}\,n * x\,\hat{}\,m) \\ \Rightarrow (x * x\,\hat{}\,n) * x\,\hat{}\,m \\ \Rightarrow x\,\hat{}\,(n+1) * x\,\hat{}\,m \end{array}$$

and we are done.

Or are we? What if, in the definition of (ˆ), x or n is *negative*? Since a negative integer is not a natural number, we could dispense with the problem by saying that these situations fall beyond the bounds of the property we are trying to prove. But let's look a little closer. If x is negative, the property we are trying to prove still holds (why?). But if n is negative, $x\,\hat{}\,n$ will not terminate (why?). As diligent programmers we may wish to defend against the latter situation by writing:

```
(ˆ) :: Integer → Integer → Integer
xˆ0 = 1
xˆn | n < 0 = error "negative exponent"
    | otherwise = x ∗ xˆ(n − 1)
```

If we consider non-terminating computations and ones that produce an error to both have the same value, namely *botom*, then these two versions of (ˆ) are equivalent. Pragmatically, however, the latter is clearly superior.

Note that the above definition will test for $n < 0$ on every recursive call, when actually the only call in which it could happen is the first. Therefore a slightly more efficient version of this program would be:

```
(ˆ) :: Integer → Integer → Integer
xˆn | n < 0 = error "negative exponent"
    | otherwise = f x n
  where f x 0 = 1
        f x n = x ∗ f x (n − 1)
```

Proving the property stated earlier for this version of the program is straightforward, with one minor distinction: what we really need to prove is that the property is true for f; that is:

$$(\forall x, n \geqslant 0, m \geqslant 0)\ \ f\ x\ (n+m) = f\ x\ n * f\ x\ m$$

from which the proof for the whole function follows trivially.

8.5.1 A More Efficient Exponentiation Function

But in fact there is a more serious inefficiency in our exponentiation function: we are not taking advantage of the fact that, for any even number n, $x^n = (x * x)^{n/2}$. Using this fact, here is a more clever way to accomplish the exponentiation task, using the names (ˆ!) and *ff* for our functions to distinguish them from the previous versions:

```
(ˆ!) :: Integer → Integer → Integer
x ˆ! n | n < 0 = error "negative exponent"
       | otherwise = ff x n
  where ff x n | n == 0 = 1
               | even n = ff (x ∗ x) (n ‘quot‘ 2)
               | otherwise = x ∗ ff x (n − 1)
```

> **Details:** *quot* is Haskell's *quotient* operator, which returns the integer quotient of the first argument divided by the second, rounded toward zero.

You should convince yourself that, intuitively at least, this version of exponentiation is not only correct, but also more efficient. More precisely, (ˆ) executes a number of steps proportional to n, whereas (ˆ!) executes a number of steps proportional to the $\log_2$ of n. The Standard Prelude defines (ˆ) similarly to the way in which (ˆ!) is defined here.

Since intuition is not always reliable, let's *prove* that this version is equivalent to the old. That is, we wish to prove that $x\,\hat{}\,n = x\ \hat{}!\ n$ for all x and n.

A quick look at the two definitions reveals that what we really need to prove is that $f\ x\ n = ff\ x\ n$, from which it follows immediately that $x\,\hat{}\,n = x\ \hat{}!\ n$. We do this by induction on n, beginning with the base case $n = 0$:

$$f\ x\ 0 \Rightarrow 1 \Rightarrow ff\ x\ 0$$

so the base step holds trivially. The induction step, however, is considerably more complicated. We must consider two cases: $n + 1$ is either even, or it is odd. If it is odd, we can show that:

$$\begin{array}{l} f\ x\ (n+1) \\ \Rightarrow x * f\ x\ n \\ \Rightarrow x * ff\ x\ n \\ \Rightarrow ff\ x\ (n+1) \end{array}$$

and we are done (note the use of the induction hypothesis in the second step).

If $n+1$ is even, we might try proceeding in a similar way:

$$\begin{array}{l} f\ x\ (n+1) \\ \Rightarrow x * f\ x\ n \\ \Rightarrow x * \mathit{ff}\ x\ n \end{array}$$

But now what shall we do? Since n is odd, we might try unfolding the call to ff:

$$\begin{array}{l} x * \mathit{ff}\ x\ n \\ \Rightarrow x * (x * \mathit{ff}\ x\ (n-1)) \end{array}$$

but this doesn't seem to be getting us anywhere. Furthermore, *folding* the call to ff (as we did in the odd case) would involve *doubling* n and taking the square root of x, neither of which seems like a good idea!

We could also try going in the other direction:

$$\begin{array}{l} \mathit{ff}\ x\ (n+1) \\ \Rightarrow \mathit{ff}\ (x*x)\ ((n+1)\ \text{`}\mathit{quot}\text{`}\ 2) \\ \Rightarrow f\ (x*x)\ ((n+1)\ \text{`}\mathit{quot}\text{`}\ 2) \end{array}$$

The use of the induction hypothesis in the second step needs to be justified, because the first argument to f has changed from x to $x*x$. But recall that the induction hypothesis states that for *all* values x, and all natural numbers up to n, $f\ x\ n$ is the same as $\mathit{ff}\ x\ n$. So this is OK.

But even allowing this, we seem to be stuck again!

Instead of pushing this line of reasoning further, let's pursue a different tact based on the (valid) assumption that if m is even, then:

$$m = m\ \text{`}\mathit{quot}\text{`}\ 2 + m\ \text{`}\mathit{quot}\text{`}\ 2$$

Let's use this fact together with the property that we proved in the last section:

$$\begin{array}{l} f\ x\ (n+1) \\ \Rightarrow f\ x\ ((n+1)\ \text{`}\mathit{quot}\text{`}\ 2 + (n+1)\ \text{`}\mathit{quot}\text{`}\ 2) \\ \Rightarrow f\ x\ ((n+1)\ \text{`}\mathit{quot}\text{`}\ 2) * f\ x\ ((n+1)\ \text{`}\mathit{quot}\text{`}\ 2) \end{array}$$

Next, as with the proof in the last section involving *reverse*, let's make an assumption about a property that will help us along. Specifically, what if we could prove that $f\ x\ n * f\ x\ n$ is equal to $f\ (x*x)\ n$? If so, we could proceed as follows:

Base case ($n = 0$):

$$\begin{array}{l} f\ x\ 0 * f\ x\ 0 \\ \Rightarrow 1 * 1 \\ \Rightarrow 1 \\ \Rightarrow f\ (x * x)\ 0 \end{array}$$

Induction step ($n + 1$):

$$\begin{array}{l} f\ x\ (n+1) * f\ x\ (n+1) \\ \Rightarrow (x * f\ x\ n) * (x * f\ x\ n) \\ \Rightarrow (x * x) * (f\ x\ n * f\ x\ n) \\ \Rightarrow (x * x) * f\ (x * x)\ n \\ \Rightarrow f\ (x * x)\ (n+1) \end{array}$$

Figure 8.1: Proof that $f\ x\ n * f\ x\ n = f\ (x * x)\ n$.

$$\begin{array}{l} f\ x\ ((n+1)\ \text{`}quot\text{`}\ 2) * f\ x\ ((n+1)\ \text{`}quot\text{`}\ 2) \\ \Rightarrow f\ (x * x)\ ((n+1)\ \text{`}quot\text{`}\ 2) \\ \Rightarrow \mathit{ff}\ (x * x)\ ((n+1)\ \text{`}quot\text{`}\ 2) \\ \Rightarrow \mathit{ff}\ x\ (n+1) \end{array}$$

and we are finally done. Note the use of the induction hypothesis in the second step, as justified earlier. The proof of the auxiliary property is not difficult, but also requires induction; it is shown in Figure 8.1.

Aside from improving efficiency, one of the pleasant outcomes of proving that (ˆ) and (ˆ!) are equivalent is that *anything that we prove about one function will be true for the other.* For example, the validity of the property that we proved earlier:

$$x\,\hat{}\,(n + m) = x\,\hat{}\,n * x\,\hat{}\,m$$

immediately implies the validity of:

$$x\ \hat{}!\ (n + m) = x\ \hat{}!\ n * x\ \hat{}!\ m$$

Although (ˆ!) is more efficient than (ˆ), it is also more complicated, so it makes sense to try proving new properties for (ˆ), since the proofs will likely be easier.

The moral of this story is that you shouldn't throw away old code that is simpler but less efficient than a newer version. That old code can serve at least two good purposes: First, if it is simpler, it is likely to be easier to understand, and thus serves a useful role in documenting your effort. Second, as we have just discussed, if it is provably equivalent to the new code, then it can be used to simplify the task of proving properties about the new code.

Exercise 8.4 The function (ˆ!) can be made more efficient by noting that in the last line of the definition of *ff*, n is odd, and therefore $n - 1$ must be even, so the test for n being even on the next recursive call could be avoided. Redefine (ˆ!) so that it avoids this (minor) inefficiency.

Exercise 8.5 Consider this definition of the *factorial* function:[3]

$$\begin{array}{l}
\mathit{fac1} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{fac1}\ 0 = 1 \\
\mathit{fac1}\ n = n * \mathit{fac1}\ (n - 1)
\end{array}$$

and this alternative definition:

$$\begin{array}{l}
\mathit{fac2} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{fac2}\ n = \mathit{fac}'\ n\ 1 \\
\quad \textbf{where}\ \mathit{fac}'\ 0\ x = x \\
\qquad\qquad\ \mathit{fac}'\ n\ x = \mathit{fac}'\ (n - 1)\ (n * x)
\end{array}$$

Prove that $\mathit{fac1}\ n = \mathit{fac2}\ n$ for all non-negative integers n.

[3] The factorial function is defined mathematically as:

$$\mathit{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \mathit{factorial}(n-1) & \text{otherwise} \end{cases}$$

Chapter 9

An Algebra of Music

In this chapter we will explore a number of properties of the *Music* data type and functions defined on it, properties that collectively form an *algebra of music*. With this algebra we can reason about, transform, and optimize computer music programs in a meaning preserving way.

9.1 Musical Equivalance

Suppose we have two values *m1* :: *Music Pitch* and *m2* :: *Music Pitch*, and we want to know if they are equal. If we treat them simply as Haskell values, we could easily write a function that compares their structures recursively to see if they are the same at every level, all the way down to the *Primitive* rests and notes. This is in fact what the Haskell function (==) does. For example, if:

```
m  = c 4 en :+: c 5 en
m1 = m :+: m
m2 = revM (revM m :+: revM m)
```

Then *m1* == *m2* is *True*.

Unfortunately, this is not always good enough from a musical point of view. For example, we would expect the following two musical values to *sound* the same, regardless of the actual values of *m1*, *m2*, and *m3*:

```
(m1 :+: m2) :+: m3
m1 :+: (m2 :+: m3)
```

In other words, we expect the operator (:+:) to be *associative*.

The problem is that, as data structures, these two values are *not* equal in general, in fact there are no finite values that can be assigned to $m1$, $m2$, and $m3$ to make them equal.[1]

The obvious way out of this dilemma is to define a new notion of equality that captures the fact that the *performances* are the same—i.e. if two things sound the same, they must be musically equivalent. And thus we define a formal notion of musical equivalence:

Definition: Two musical values $m1$ and $m2$ are *equivalent*, written $m1 \equiv m2$, if and only if:

$$(\forall pmap, c)\ \ perf\ pmap\ c\ m1 = perf\ pmap\ c\ m2$$

We will study a number of properties in this chapter that capture musical equivalences, similar in spirit to the associativity of (:+:) above. Each of them can be thought of as an *axiom*, and the set of valid axioms collectively forms an *algebra of music*. By proving the validity of each axiom we not only confirm our intuitions about how music is interpreted, but also gain confidence that our *perform* function actually does the right thing. Furthermore, with these axioms in hand, we can *transform* musical values in meaning-preserving ways.

Speaking of the *perform* function, recall from Chapter 6 that we defined *two* versions of *perform*, and the definition above uses the function *perf*, which includes the duration of a musical value in its result. The following Lemma captures the connection between these functions:

Lemma 9.1.1 For all *pmap*, *c*, and *m2*:

$$perf\ pmap\ c\ m2 = (perform\ pmap\ c\ m2, dur\ m2)$$

where *perform* is the function defined in Figure 6.1.

To see the importance of including duration in the definition of equivalence, we first note that if two musical values are equivalent, we should be able to substitute one for the other in any valid musical context. But if duration is not taken into account, then all rests are equivalent (because their performances are just the empty list). This means that, for example, $m1 :\!+\!: rest\ 1 :\!+\!: m2$ is equivalent to $m1 :\!+\!: rest\ 2 :\!+\!: m2$, which is surely not what we want.

[1] If $m1 = m1 :\!+\!: m2$ and $m3 = m2 :\!+\!: m3$ then the two expressions are equal, but these are infinite values that cannot even be performed.

Note that we could have defined *perf* as above, i.e. in terms of *perform* and *dur*, but as mentioned in Section 6.1 it would have been computationally inefficient to do so. On the other hand, if the Lemma above is true, then our proofs might be simpler if we first proved the property using *perform*, and then using *dur*. That is, to prove $m1 \equiv m2$ we need to prove:

$$perf\ pmap\ c\ m1 = perf\ pmap\ c\ m2$$

Instead of doing this directly using the definition of *perf*, we could instead prove both of the following:

$$perform\ pmap\ c\ m1 = perform\ pmap\ c\ m2$$
$$dur\ m1 = dur\ m2$$

9.2 Some Simple Axioms

Let's look at a few simpler axioms, and see how we can prove each of them using the proof techniques that we have developed so far.

Axiom 9.2.1 For any *r1*, *r2*, and *m*:

$$Modify\ (Tempo\ r1)\ (Modify\ (Tempo\ r2)\ m) \equiv Modify\ (Tempo\ (r1 * r2))\ m$$

In other words, *tempo scaling is multiplicative.*

We can prove this by calculation, starting with the definition of musical equivalence. For clarity we will first prove the property for *perform*, and then for *dur*, as suggested in the last section:

let $dt = cDur\ c$

$perform\ pmap\ c\ (Modify\ (Tempo\ r1)\ (Modify\ (Tempo\ r2)\ m))$
$\Rightarrow \{unfold\ perform\}$
$perform\ pmap\ (c\{\ cDur = dt\ /\ r1\ \})\ (Modify\ (Tempo\ r2)\ m)$
$\Rightarrow \{unfold\ perform\}$
$perform\ pmap\ (c\{\ cDur = (dt\ /\ r1)\ /\ r2\ \})\ m$
$\Rightarrow \{arithmetic\}$
$perform\ pmap\ (c\{\ cDur = dt\ /\ (r1 * r2)\ \})\ m$
$\Rightarrow \{fold\ perform\}$
$perform\ pmap\ c\ (Modify\ (Tempo\ (r1 * r2))\ m)$

$dur\ (Modify\ (Tempo\ r1)\ (Modify\ (Tempo\ r2)\ m))$
$\Rightarrow \{unfold\ dur\}$

$$\begin{array}{l}
\mathit{dur}\ (\mathit{Modify}\ (\mathit{Tempo}\ \mathit{r2})\ m)\ /\ \mathit{r1}\\
\Rightarrow \{\mathit{unfold}\ \mathit{dur}\}\\
(\mathit{dur}\ m\ /\ \mathit{r2})\ /\ \mathit{r1}\\
\Rightarrow \{\mathit{arithmetic}\}\\
\mathit{dur}\ m\ /\ (\mathit{r1} * \mathit{r2})\\
\Rightarrow \{\mathit{fold}\ \mathit{dur}\}\\
\mathit{dur}\ (\mathit{Modify}\ (\mathit{Tempo}\ (\mathit{r1} * \mathit{r2}))\ m)
\end{array}$$

Here is another useful axiom and its proof:

Axiom 9.2.2 For any r, $\mathit{m1}$, and $\mathit{m2}$:

$$\mathit{Modify}\ (\mathit{Tempo}\ r)\ (\mathit{m1}\ {:}{+}{:}\ \mathit{m2}) \equiv \mathit{Modiy}\ (\mathit{Tempo}\ r)\ \mathit{m1}\ {:}{+}{:}\ \mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m2}$$

In other words, *tempo scaling distributes over sequential composition.*

Proof:

$$\begin{array}{l}
\textbf{let}\ t = \mathit{cTime}\ c; \mathit{dt} = \mathit{cDur}\ c\\
\quad\ \ \mathit{t1} = t + \mathit{dur}\ \mathit{m1} * (\mathit{dt}\ /\ r)\\
\quad\ \ \mathit{t2} = t + (\mathit{dur}\ \mathit{m1}\ /\ r) * \mathit{dt}\\
\quad\ \ \mathit{t3} = t + \mathit{dur}\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m1}) * \mathit{dt}
\end{array}$$

$$\begin{array}{l}
\mathit{perform}\ \mathit{pmap}\ c\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ (\mathit{m1}\ {:}{+}{:}\ \mathit{m2}))\\
\Rightarrow \{\mathit{unfold}\ \mathit{perform}\}\\
\mathit{perform}\ \mathit{pmap}\ (c\{\mathit{cDur} = \mathit{dt}\ /\ r\})\ (\mathit{m1}\ {:}{+}{:}\ \mathit{m2})\\
\Rightarrow \{\mathit{unfold}\ \mathit{perform}\}\\
\mathit{perform}\ \mathit{pmap}\ (c\{\mathit{cDur} = \mathit{dt}\ /\ r\})\ \mathit{m1}\\
\quad +\!\!+ \mathit{perform}\ \mathit{pmap}\ (c\{\mathit{cTime} = \mathit{t1}, \mathit{cDur} = \mathit{dt}\ /\ r\})\ \mathit{m2}\\
\Rightarrow \{\mathit{fold}\ \mathit{perform}\}\\
\mathit{perform}\ \mathit{pmap}\ c\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m1})\\
\quad +\!\!+ \mathit{perform}\ \mathit{pmap}\ (c\{\mathit{cTime} = \mathit{t1}\})\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m2})\\
\Rightarrow \{\mathit{arithmetic}\}\\
\mathit{perform}\ \mathit{pmap}\ c\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m1})\\
\quad +\!\!+ \mathit{perform}\ \mathit{pmap}\ (c\{\mathit{cTime} = \mathit{t2}\})\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m2})\\
\Rightarrow \{\mathit{fold}\ \mathit{dur}\}\\
\mathit{perform}\ \mathit{pmap}\ c\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m1})\\
\quad +\!\!+ \mathit{perform}\ \mathit{pmap}\ (c\{\mathit{cTime} = \mathit{t3}\})\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m2})\\
\Rightarrow \{\mathit{fold}\ \mathit{perform}\}\\
\mathit{perform}\ \mathit{pmap}\ c\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m1}\ {:}{+}{:}\ \mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m2})
\end{array}$$

$$\mathit{dur}\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ (\mathit{m1}\ {:}{+}{:}\ \mathit{m2}))$$

⇒ *dur* (*m1* :+: *m2*) / *r*
⇒ (*dur m1* + *dur m2*) / *r*
⇒ *dur m1* / *r* + *dur m2* / *r*
⇒ *dur* (*Modify* (*Tempo r*) *m1*) + *dur* (*Modify* (*Tempo r*) *m2*)
⇒ *dur* (*Modify* (*Tempo r*) *m1* :+: *Modify* (*Tempo r*) *m2*)

An even simpler axiom is given by:

Axiom 9.2.3 For any m, *Modify* (*Tempo* 1) $m \equiv m$.

In other words, *unit tempo scaling is the identity function for type Music.*

Proof:

let *dt* = *cDur c*

perform pmap c (*Modify* (*Tempo* 1) *m*)
⇒ {*unfold perform*}
perform pmap (*c*{*cDur* = *dt* / 1}) *m*
⇒ {*arithmetic*}
perform pmap c m

dur (*Modify* (*Tempo* 1) *m*)
⇒ *dur m* / 1
⇒ *dur m*

Note that the above three proofs, being used to establish axioms, all involve the definition of *perform*. In contrast, we can also establish *theorems* whose proofs involve only the axioms. For example, Axioms 1, 2, and 3 are all needed to prove the following:

Theorem 9.2.1 For any r, *m1*, and *m2*:

Modify (*Tempo r*) *m1* :+: *m2* ≡ *Modify* (*Tempo r*) (*m1* :+: *Modify* (*Tempo* (1 / *r*)) *m2*)

Proof:

Modify (*Tempo r*) *m1* :+: *m2*
⇒ {*Axiom* 3}
Modify (*Tempo r*) *m1* :+: *Modify* (*Tempo* 1) *m2*
⇒ {*arithmetic*}
Modify (*Tempo r*) *m1* :+: *Modify* (*Tempo* (*r* ∗ (1 / *r*))) *m2*

$$\begin{array}{l}
\Rightarrow \{\mathit{Axiom}\ 1\} \\
\mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m1}\ {:+:}\ \mathit{Modify}\ (\mathit{Tempo}\ r)\ (\mathit{Modify}\ (\mathit{Tempo}\ (1\ /\ r))\ \mathit{m2}) \\
\Rightarrow \{\mathit{Axiom}\ 2\} \\
\mathit{Modify}\ (\mathit{Tempo}\ r)\ (\mathit{m1}\ {:+:}\ \mathit{Modify}\ (\mathit{Tempo}\ (1\ /\ r))\ \mathit{m2})
\end{array}$$

9.3 The Axiom Set

There are many other useful axioms, but we do not have room to include all of their proofs here. They are listed below, which include the axioms from the previous section as special cases, and the proofs are left as exercises.

Axiom 9.3.1 *Tempo* is *multiplicative* and *Transpose* is *additive.* That is, for any $\mathit{r1}$, $\mathit{r2}$, $\mathit{p1}$, $\mathit{p2}$, and m:

$$\begin{array}{l}
\mathit{Modify}\ (\mathit{Tempo}\ \mathit{r1})\ (\mathit{Modify}\ (\mathit{Tempo}\ \mathit{r2})\ m) \equiv \mathit{Modify}\ (\mathit{Tempo}\ (\mathit{r1} * \mathit{r2}))\ m \\
\mathit{Modify}\ (\mathit{Trans}\ \mathit{p1})\ (\mathit{Modify}\ (\mathit{Trans}\ \mathit{p2})\ m) \equiv \mathit{Modify}\ (\mathit{Trans}\ (\mathit{p1} + \mathit{p2}))\ m
\end{array}$$

Axiom 9.3.2 Function composition is *commutative* with respect to both tempo scaling and transposition. That is, for any $\mathit{r1}$, $\mathit{r2}$, $\mathit{p1}$ and $\mathit{p2}$:

$$\begin{array}{l}
\mathit{Modify}\ (\mathit{Tempo}\ \mathit{r1}) \circ \mathit{Modify}\ (\mathit{Tempo}\ \mathit{r2}) \equiv \mathit{Modify}\ (\mathit{Tempo}\ \mathit{r2}) \circ \mathit{Modify}\ (\mathit{Tempo}\ \mathit{r1}) \\
\mathit{Modify}\ (\mathit{Trans}\ \mathit{p1}) \circ \mathit{Modify}\ (\mathit{Trans}\ \mathit{p2}) \equiv \mathit{Modify}\ (\mathit{Trans}\ \mathit{p2}) \circ \mathit{Modify}\ (\mathit{Trans}\ \mathit{p1}) \\
\mathit{Modify}\ (\mathit{Tempo}\ \mathit{r1}) \circ \mathit{Modify}\ (\mathit{Trans}\ \mathit{p1}) \equiv \mathit{Modify}\ (\mathit{Trans}\ \mathit{p1}) \circ \mathit{Modify}\ (\mathit{Tempo}\ \mathit{r1})
\end{array}$$

Axiom 9.3.3 Tempo scaling and transposition are *distributive* over both sequential and parallel composition. That is, for any r, p, $\mathit{m1}$, and $\mathit{m2}$:

$$\begin{array}{l}
\mathit{Modify}\ (\mathit{Tempo}\ r)\ (\mathit{m1}\ {:+:}\ \mathit{m2}) \equiv \mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m1}\ {:+:}\ \mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m2} \\
\mathit{Modify}\ (\mathit{Tempo}\ r)\ (\mathit{m1}\ {:=:}\ \mathit{m2}) \equiv \mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m1}\ {:=:}\ \mathit{Modify}\ (\mathit{Tempo}\ r)\ \mathit{m2} \\
\mathit{Modify}\ (\mathit{Trans}\ p)\ (\mathit{m1}\ {:+:}\ \mathit{m2}) \equiv \mathit{Modify}\ (\mathit{Trans}\ p)\ \mathit{m1}\ {:+:}\ \mathit{Modify}\ (\mathit{Trans}\ p)\ \mathit{m2} \\
\mathit{Modify}\ (\mathit{Trans}\ p)\ (\mathit{m1}\ {:=:}\ \mathit{m2}) \equiv \mathit{Modify}\ (\mathit{Trans}\ p)\ \mathit{m1}\ {:=:}\ \mathit{Modify}\ (\mathit{Trans}\ p)\ \mathit{m2}
\end{array}$$

Axiom 9.3.4 Sequential and parallel composition are *associative.* That is, for any $\mathit{m0}$, $\mathit{m1}$, and $\mathit{m2}$:

$$\begin{array}{l}
\mathit{m0}\ {:+:}\ (\mathit{m1}\ {:+:}\ \mathit{m2}) \equiv (\mathit{m0}\ {:+:}\ \mathit{m1})\ {:+:}\ \mathit{m2} \\
\mathit{m0}\ {:=:}\ (\mathit{m1}\ {:=:}\ \mathit{m2}) \equiv (\mathit{m0}\ {:=:}\ \mathit{m1})\ {:=:}\ \mathit{m2}
\end{array}$$

Axiom 9.3.5 Parallel composition is *commutative.* That is, for any $\mathit{m0}$ and $\mathit{m1}$:

$$\mathit{m0}\ {:=:}\ \mathit{m1} \equiv \mathit{m1}\ {:=:}\ \mathit{m0}$$

Axiom 9.3.6 *Rest* 0 is a *unit* for *Tempo* and *Trans*, and a *zero* for sequential and parallel composition. That is, for any r, p, and m:

$$
\begin{aligned}
&\mathit{Tempo}\ r\ (\mathit{Rest}\ 0) \equiv \mathit{Rest}\ 0\\
&\mathit{Trans}\ p\ (\mathit{Rest}\ 0) \equiv \mathit{Rest}\ 0\\
&m :\!+\!: \mathit{Rest}\ 0 \equiv m \equiv \mathit{Rest}\ 0 :\!+\!: m\\
&m :\!=\!: \mathit{Rest}\ 0 \equiv m \equiv \mathit{Rest}\ 0 :\!=\!: m
\end{aligned}
$$

Axiom 9.3.7 There is a duality between (:+:) and (:+:), namely that, for any $\mathit{m0}$, $\mathit{m1}$, $\mathit{m2}$, and $\mathit{m3}$ such that $\mathit{dur}\ \mathit{m0} = \mathit{dur}\ \mathit{m2}$:

$$
(\mathit{m0} :\!+\!: \mathit{m1}) :\!=\!: (\mathit{m2} :\!+\!: \mathit{m3}) \equiv (\mathit{m0} :\!=\!: \mathit{m2}) :\!+\!: (\mathit{m1} :\!=\!: \mathit{m3})
$$

Exercise 9.1 Establish the validity of each of the above axioms.

Exercise 9.2 Recall the function *revM* defined in Chapter 2, and note that, in general, $\mathit{revM}\ (\mathit{revM}\ m)$ is not equal to m. However, the following is true:

$$
\mathit{revM}\ (\mathit{revM}\ m) \equiv m
$$

Prove this fact by calculation.

9.4 Soundness and Completeness

TBD

Chapter 10

Musical L-Systems

```
module Haskore.LSystems where
import Data.List
import System.Random
import Haskore
```

10.1 Generative Grammars

A *grammar* describes a *formal language*. One can either design a *recognizer* (or *parser*) for that language, or design a *generator* that generates sentences in that language. We are interested in using grammars to generate music, and thus we are only interested in generative grammars.

A generative grammar is a four-tuple (N, T, n, P), where:

- N is the set of *non-terminal symbols*.
- T is the set of *terminal symbols*.
- n is the *initial symbol*.
- P is a set of *production rules*, where each production rule is a pair (X, Y), often written $X \to Y$, where X and Y are words over the alphabet $N \cup T$, and X contains at least one non-terminal.

A *Lindenmayer system*, or *L-system*, is an example of a generative grammer, but is different in two ways:

1. The *sequence* of sentences is as important as the individual sentences, and

2. A new sentence is generated from the previous one by applying as many productions as possible on each step—a kind of "parallel production."

Lindenmayer was a biologist and mathematician, and he used L-systems to describe the growth of certain biological organisms (such as plants, and in particular algae).

We will limit our discussion to L-systems that have the following additional characteristics:

1. They are *context-free*: the left-hand side of each production (i.e. X above) is a single non-terminal.

2. No distinction is made between terminals and non-terminals (with no loss of expressive power—why?).

We will consider both *deterministic* and *non-deterministic* grammars. A deterministic grammar has exactly one production corresponding to each terminal symbol in the alphabet, whereas a non-deterministic grammar may have more than one, and thus we will need some way to choose between them.

10.2 A Simple Implementation

A very simple context-free, deterministic grammar can be designed as follows. We represent the set of productions as a list of symbol/list-of-symbol pairs:

```
data DetGrammar a = DetGrammar a            -- start symbol
                                 [(a, [a])]    -- productions
  deriving Show
```

To generate a succession of "sentential forms," we need to define a function that, given a grammar, returns a list of lists of symbols:

```
detGenerate :: Eq a ⇒ DetGrammar a → [[a]]
detGenerate (DetGrammar st ps) = iterate (concatMap f) [st]
            where f a = maybe [a] id (lookup a ps)
```

Note that we will each symbol "in parallel" at each step, using *concatMap*. The repetition of this process at each step is achieved using *iterate*. Note also that a list of productions is essentially an *association list*, and thus the library function *lookup* works quite well in finding the production rule that we seek. Finally, note once again how the use of higher-order functions makes this definition concise yet efficient.

As an example of the use of this simple program, a Lindenmayer grammer for red algae (taken from []) is given by:

```
redAlgae = DetGrammar 'a'
                [('a',"b|c"),
                 ('b',"b"),
                 ('c',"b|d"),
                 ('d',"e\\d"),
                 ('e',"f"),
                 ('f',"g"),
                 ('g',"h(a)"),
                 ('h',"h"),
                 ('|',"|"),
                 ('(',"("),
                 (')',")"),
                 ('/',"\\"),
                 ('\\',"/")
                ]
```

Then *detGenerate redAlgae* gives us the result that we want—or, to make it look nicer, we could do:

```
t n g = sequence_ (map putStrLn (take n (detGenerate g)))
```

For example, the 10th element of *t* 10 *redAlgae* is:

```
"b|b|h(b|b|e\d)\h(b|b|d)/h(b|c)\h(a)/g\f/e\d"
```

Exercise 10.1 Design a function *testDet* :: *Grammar a* → *Bool* such that *testDet g* is *True* if *g* has exactly one rule for each of its symbols; i.e. it is deterministic. Then modify the *generate* function above so that it returns an error if a grammer not satisfying this constraint is given as argument.

10.3 Grammars in Haskell

The design given in the last section only captures deterministic context-free grammars. We would also like to consider non-deterministic grammars, where a user can specify the probability that a particular rule is selected, as well as possibly non-context free (i.e. context sensitive) grammars. Thus we will represent a generative grammar a bit more abstractly, as a data structure that has a starting sentence in an (implicit, polymorphic) alphabet, and a list of production rules:

```
data Grammar a = Grammar a        -- start sentence
                         (Rules a)       -- production rules
     deriving Show
```

The production rules are instructions for converting sentences in the alphabet to other sentences in the alphabet. A rule set is either a set of uniformly distributed rules (meaning that those with the same left-hand side have an equal probability of being chosen), or a set of stochastic rules (each of which is paired with a probabilty). A specific rule consists of a left-hand side and a right-hand side.

```
data Rules a = Uni [Rule a]
             | Sto [(Rule a, Prob)]
     deriving (Eq, Ord, Show)

data Rule a = Rule{ lhs :: a, rhs :: a}
     deriving (Eq, Ord, Show)

type Prob = Float
```

One of the key sub-problems that we will have to solve is how to probabilistically select a rule from a set of rules, and use that rule to expand a non-terminal. We define the following type to capture this process:

```
type ReplFun a = [[(Rule a, Prob)]] → (a, [Rand]) → (a, [Rand])
type Rand = Float
```

The idea is that a function f :: *ReplFun a* is such that f *rules* (s, *rands*) will return a new sentence s' in which each symbol in s has been replaced according to some rule in *rules* (which are grouped by common left-hand side). Each rule is chosen probabilitically based on the random numbers in *rands*, and thus the result also includes a new list of random numbers to account for those "consumed" by the replacement process.

With such a function in hand, we can now define a function that, given a grammar, generates an infinite list of the sentences produced by this replacement process. Because the process is non-deterministic, we also pass a seed (an integer) to generate the initial pseudo-random number sequence to give us repeatable results.

```
gen :: Ord a ⇒ ReplFun a → Grammar a → Int → [a]
gen f (Grammar s rules) seed =
    let Sto newRules = toStoRules rules
```

```
          rands = randomRs (0.0, 1.0) (mkStdGen seed)
      in if checkProbs newRules
         then generate f newRules (s, rands)
         else (error "Stochastic rule-set is malformed.")
```

toStoRules converts a list of uniformly distributed rules to an equivalent list of stochastic rules. Each set of uniform rules with the same LHS is converted to a set of stochastic rules in which the probability of each rule is one over the number of uniform rules.

```
toStoRules :: (Ord a, Eq a) ⇒ Rules a → Rules a
toStoRules (Sto rs) = Sto rs
toStoRules (Uni rs) =
  let rs′ = groupBy (λr1 r2 → lhs r1 == lhs r2) (sort rs)
  in Sto (concatMap insertProb rs′)

insertProb :: [a] → [(a, Prob)]
insertProb rules = let prb = 1.0 / fromIntegral (length rules)
                   in zip rules (repeat prb)
```

checkProbs takes a list of production rules and checks whether, for every rule with the same LHS, the probabilities sum to one (plus or minus some epsilon, currenty set to 0.001).

```
checkProbs :: (Ord a, Eq a) ⇒ [(Rule a, Prob)] → Bool
checkProbs rs = and (map checkSum (groupBy sameLHS (sort rs)))

eps = 0.001

checkSum :: [(Rule a, Prob)] → Bool
checkSum rules = let mySum = sum (map snd rules)
                 in abs (1.0 − mySum) ⩽ eps

sameLHS :: Eq a ⇒ (Rule a, Prob) → (Rule a, Prob) → Bool
sameLHS (r1, f1) (r2, f2) = lhs r1 == lhs r2
```

generate takes a list of rules, a replacement function, a starting sentence, and a source of random numbers. It returns an infinite list of sentences.

```
generate :: Eq a ⇒ ReplFun a → [(Rule a, Prob)] → (a, [Rand]) → [a]
generate f rules xs =
  let newRules = map probDist (groupBy sameLHS rules)
```

```
      probDist rrs = let (rs, ps) = unzip rrs
                     in zip rs (tail (scanl (+) 0 ps))
  in map fst (iterate (f newRules) xs)
```

10.4 An L-System Grammar for Music

The above is all for a generic grammar. For a musical L-system we will define a specific grammar, whose sentences are defined as follows. A musical L-system sentence is either:

- A non-terminal symbol (*N a*).
- A sequential composition *s1* :+ *s2*.
- A functional composition *s1* :. *s2*.
- The symbol *Id*, which will eventually interpeted as the identity function.

We capture this in the *LSys* data type:

```
data LSys a = N a
            | LSys a :+ LSys a
            | LSys a :. LSys a
            | Id
     deriving (Eq, Ord, Show)
```

We also need to define a replacement function for this grammar. We treat (:+) and (:.) as binary branches, and recursively traverse each of their arguments. We treat *Id* as a constant that never gets replaced. Most importantly, each non-terminal of the form *N x* could each be the left-hand side of a rule, so we call the function *getNewRHS* to generate the replace term for it.

```
replFun :: Eq a ⇒ ReplFun (LSys a)
replFun rules (s, rands) =
  case s of
    a :+ b → let (a′, rands′) = replFun rules (a, rands)
                 (b′, rands″) = replFun rules (b, rands′)
             in (a′ :+ b′, rands″)
    a :. b → let (a′, rands′) = replFun rules (a, rands)
                 (b′, rands″) = replFun rules (b, rands′)
```

```
              in (a′ :. b′, rands″)
          Id → (Id, rands)
          N x → (getNewRHS rules (N x) (head rands), tail rands)
```

Note the use of *filter* to select only the rules whose left-hand side matches the non-terminal. A key aspect of the algorithm is to then generate the *probability density* of the successive rules, which is basically the sum of its probability plus the probabilities of all rules that precede it. This modified rule-set is then given to *getNewRHS* as an argument. *getNewRHS* is defined as:

```
getNewRHS :: Eq a ⇒ [[(Rule a, Prob)]] → a → Rand → a
getNewRHS rrs ls rand =
  let loop ((r, p) : rs) = if rand ⩽ p then rhs r else loop rs
      loop [ ] = error "getNewRHS anomaly"
  in case (find (λ((r, p) : _) → lhs r == ls) rrs) of
        Just rs → loop rs
        Nothing → error "No rule match"
```

10.5 Examples

The final step is to interpret the resulting sentence (i.e. a value of type *LSys a*) as music. The intent of the *LSys* design is that a value is interpreted as a *function* that is applied to a single note (or, more generally, a single *Music* value). The specific constructors are interpreted as follows:

```
type IR a b = [(a, Music b → Music b)]      -- IR stands for interpetation rules

interpret :: (Eq a) ⇒ LSys a → IR a b → Music b → Music b
interpret (a :. b) r m = interpret a r (interpret b r m)
interpret (a :+ b) r m = interpret a r m :+: interpret b r m
interpret Id r m = m
interpret (N x) r m = case (lookup x r) of
                           Just f → f m
                           Nothing → error "No interpetation rule"
```

For example, we could define the following interpretation rules:

```
data LFun = Inc | Dec | Same
     deriving (Eq, Ord, Show)
```

```
ir :: IR LFun Pitch
ir = [(Inc, Haskore.transpose 1),
      (Dec, Haskore.transpose (−1)),
      (Same, id)]

inc, dec, same :: LSys LFun
inc = N Inc
dec = N Dec
same = N Same
```

In other words, *inc* transposes the music up by one semitone, *dec* transposes it down by a semitone, and *same* does nothing.

Now let's build an actual grammar. *sc* increments a note followed by its decrement – the two notes are one whole tone apart:

```
sc = inc :+ dec
```

Now let's define a bunch of rules as follows:

```
r1a = Rule inc (sc :. sc)
r1b = Rule inc sc
r2a = Rule dec (sc :. sc)
r2b = Rule dec sc
r3a = Rule same inc
r3b = Rule same dec
r3c = Rule same same
```

and the corresponding grammar:

```
g1 = Grammar same (Uni [r1b, r1a, r2b, r2a, r3a, r3b])
```

Finally, we generate a sentence at some particular level, and interpret it as music:

```
t1 n = instrument Vibraphone $
         interpret (gen replFun g1 42 !! n) ir (c 5 tn)
```

Try "*play t1* 3" or "*play t1* 4" to hear the result.

Exercise 10.2 Play with the L-System grammar defined above. Change the production rules. Add probabilities to the rules, i.e. change it into a *Sto* grammar. Change the random number seed. Change the depth of recursion. And also try changing the "musical seed" (i.e. the note *c* 5 *tn*).

Exercise 10.3 Define a new L-System structure. In particular, (a) define a new version of *LSys* (for example, add a parallel constructor) and its associated interpretation, and/or (b) define a new version of *LFun* (perhaps add something to control the volume) and its associated interpretation. Then define some grammars with the new design to generate interesting music.

Chapter 11

Qualified Types

Recall that a polymorphic type such as $(a \rightarrow a)$ is really shorthand for $\forall(a) a \rightarrow a$, which can be read "*for all* types a, functions mapping elements of type a to elements of type a." Note the emphasis on *for all.*

In practice, however, there are times when we would prefer to limit a polymorphic type to a smaller number of possibilities. A good example is a function such as $(+)$. It's probably not a good idea to limit $(+)$ to a *single* (that is, *mono*morphic) type such as $\mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}$, since there are other kinds of numbers—such as rational and floating-point numbers—that we would like to perform addition on. Nor is it a very good idea to have a different addition function for each type of number we wish to add, since that would require giving each a different name, such as *addInteger*, *addRational*, *addFloat*, etc. And, unfortunately, we can't give $(+)$ a type such as $a \rightarrow a \rightarrow a$ since this would imply that we could add things other than numbers, such as characters, lists, tuples, and any type that you might define on your own!

Haskell provides a solution to this problem through the use of *qualified types.* Conceptually, you can think of a qualified type just as a polymorphic type, except that in place of "*for all* types a" we will be able to say "for all types *a that are members of class C*," where the class C can be thought of as a set of types. For example, suppose there is a class *Num* with members *Integer*, *Rational*, and *Float*. Then we could give an accurate type for $(+)$, namely: $\forall(a \in \mathit{Num}) a \rightarrow a \rightarrow a$. But in Haskell, instead of writing $\forall(a \in \mathit{Num}) \cdots$ we will write $\mathit{Num}\ a \Rightarrow \cdots$. So the proper type signature for $(+)$ is:

$$(+) :: \mathit{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$$

which should be read: "for all types a that are members of the class *Num*,

(+) has type $a \to a \to a$." Members of a class are also called *instances* of the class, and we will use these two terms interchangeably in the remainder of the text. The $Num\ a \Rightarrow \cdots$ part of the type signature is often called a *context*, or *constraint*.

> **Details:** It is important not to confuse Num with a data type or a constructor within a data type, even though the same syntax ("$Num\ a$") is used. Num is a *type class*, and the context of its use (namely, to the left of a $\Rightarrow$) is always sufficient to determine this fact.

The ability to qualify polymorphic types is a unique feature of Haskell, and, as you will soon see, provides great expressiveness. In particular, you will see that it is possible to define your own type class and its members. But first, let's look at another example of a pre-defined qualified type in Haskell.

11.1 Equality

Equality between two expressions *e1* and *e2* in Haskell means that the value of *e1* is the same as the value of *e2*. Another way to view equality is that you should be able to substitute *e1* for *e2* wherever they appear in a program, without affecting the result of that program.

In general, however, it is not possible for a program to determine the equality of two expressions—consider, for example, determining the equality of two infinite lists, or the equality of two functions of type $Integer \to Integer$. The ability to compute the equality of two values is called *computational equality*. Even though by the above simple examples it is clear that computational equality is strictly weaker than full equality, it is still an operation that we would like to use in many ordinary programs.

Haskell's operator for computational equality is (==). Partly because of the problem mentioned above, there are many types for which we would like equality defined, but some for which we might not. For example, we often want to compare two characters, two integers, two floating-point numbers, etc. On the other hand, comparing the equality of functions is difficult, and in general not possible. Thus Haskell has a type class called *Eq*, so that the equality operator (==) can be given the qualified type:

$$(==) :: Eq\ a \Rightarrow a \to a \to Bool$$

In other words, (==) is a function that, for any type *a* in the class *Eq*, tests two values of type *a* for equality, returning a Boolean (*Bool*) value as a result. Amongst *Eq*'s instances are the types *Char* and *Integer*, so that the following calculations hold:

$$42 == 42 \Rightarrow \mathit{True}$$
$$42 == 43 \Rightarrow \mathit{False}$$
$$\texttt{'a'} == \texttt{'a'} \Rightarrow \mathit{True}$$
$$\texttt{'a'} == \texttt{'b'} \Rightarrow \mathit{False}$$

Furthermore, the expression $42 == \texttt{'a'}$ is *ill-typed*; Haskell is clever enough to know when qualified types are ill-formed.

One of the nice things about qualified types is that they work in the presence of ordinary polymorphism. In particular, the type constraints can be made to propagate through polymorphic data types. For example, because *Integer* and *Float* are members of *Eq*, so are the types $(\mathit{Integer}, \mathit{Char})$, $[\mathit{Integer}]$, $[\mathit{Float}]$, etc. Thus:

$$[42, 43] == [42, 43] \Rightarrow \mathit{True}$$
$$[4.2, 4.3] == [4.3, 4.2] \Rightarrow \mathit{False}$$
$$(42, \texttt{'a'}) == (42, \texttt{'a'}) \Rightarrow \mathit{True}$$

We will see how this is done is a later section.

Type constraints also propagate through function definitions. For example, consider this definition of the function *elem* which tests for membership in a list:

$$x \in [\,] = \mathit{False}$$
$$x \in (y : ys) = x == y \lor x \in ys$$

Note the use of $(==)$ on the right-hand side of the second equation. The principal type for *elem* is thus:

$$\mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Bool}$$

This should be read, "For every type a that is an instance of the class *Eq*, *elem* has type $a \to [a] \to \mathit{Bool}$." This is just what we want—it expresses the fact that *elem* is not defined on all types, just those for which computational equality is defined.

The above type for *elem* is also its principal type, and Haskell will infer this type if no signature is given. Indeed, if you were to write the type signature:

$$\mathit{elem} :: a \to [a] \to \mathit{Bool}$$

you would encounter a type error, because this type is fundamentally *too general*, and the Haskell type system will complain.

> **Details:** On the other hand, you could write:
>
> $$elem :: Integer \rightarrow [Integer] \rightarrow Bool$$
>
> if you expect to use $elem$ only on lists of integers. In other words, using a type signature to constrain a value to be less general than its principal type is Ok.

As another example of this idea, a function that squares its argument:

$$square\ x = x * x$$

has principal type $Num\ a \Rightarrow a \rightarrow a$, since $(*)$, like $(+)$, has type $Num\ a \Rightarrow a \rightarrow a \rightarrow a$. Thus:

$$square\ 42 \Rightarrow 1764$$
$$square\ 4.2 \Rightarrow 17.64$$

We will study the *Num* class in greater detail shortly.

11.2 Defining Your Own Type Classes

Haskell provides a mechanism whereby you can create your own qualified types, by defining a new type class and specifying which types are members, or "instances" of it. Indeed, the type classes *Num* and *Eq* are not built-in as primitives in Haskell, but rather are simply predefined in the Standard Prelude.

To see how this is done, let's take the *Eq* class as an example. It is created by the following *type class declaration*:

$$\begin{array}{l}\textbf{class}\ Eq\ a\ \textbf{where}\\ \quad (==) :: a \rightarrow a \rightarrow Bool\end{array}$$

The connection between $(==)$ and *Eq* is important: the above declaration should be read "a type a is an instance of the class *Eq* only if there is an operation $(==) :: a \rightarrow a \rightarrow Bool$ defined on it."

> **Details:** $(==)$ is called an *operation* in the class Eq, and in general more than one operation is allowed in a class. We will see examples of this shortly.

So far so good. But how do we specify which types are instances of the class *Eq*, and the actual behavior of (==) on each of those types? This is done with an *instance declaration*. For example:

```
instance Eq Integer where
  x == y = integereq x y
```

The definition of (==) is called a *method*. The function *integerEq* happens to be the primitive function that compares integers for equality, but in general any valid expression is allowed on the right-hand side, just as for any other function definition. The overall instance declaration is essentially saying: “The type *Integer* is an instance of the class *Eq*, and here is the method corresponding to the operation (==).” Given this declaration, we can now compare fixed-precision integers for equality using (==). Similarly:

```
instance Eq Float where
  x == y = floatEq x y
```

allows us to compare floating-point numbers using (==).

More importantly, datatypes that you have defined on your own can also be made instances of the class *Eq*. Consider, for example, the *PitchClass* data type defined in Chapter 2:

```
data PitchClass = Cff | Cf | C | Dff | Cs | Df | Css | D | Eff | Ds
                | Ef | Fff | Dss | E | Es | Ff | F | Gff | Ess | Fs
                | Gf | Fss | G | Aff | Gs | Af | Gss | A | Bff | As
                | Bf | Ass | B | Bs | Bss
```

We can declare *PitchClass* to be an instance of *Eq* as follows:

```
instance Eq PitchClass where
  Cff == Cff = True
  Cf == Cf = True
  C == C = True
  ...
  Bs == Bs = True
  Bss == Bss = True
  _ == _ = False
```

where ... refers to the other 30 equations to make this definition of (==) complete. Indeed, this is rather tedious! It is not only tedious, it is also dead obvious how (==) should be defined. Therefore Haskell provides a convenient way to automatically derive such instance declarations from data type declarations, for certain type classes. In the case of *PitchClass*, all we have to do is add a **deriving** clause:

```
data PitchClass = Cff | Cf | C | Dff | Cs | Df | Css | D | Eff | Ds
                | Ef | Fff | Dss | E | Es | Ff | F | Gff | Ess | Fs
                | Gf | Fss | G | Aff | Gs | Af | Gss | A | Bff | As
                | Bf | Ass | B | Bs | Bss
     deriving Eq
```

With this declaration, Haskell will automatically derive the instance declaration that we defined above, so that (==) behaves in the way we would expect it to.

Let's now consider a polymorphic type, such as the *Primitive* type from Chapter 2:

```
data Primitive a = Note Dur a
                 | Rest Dur
```

What should an instance for this type in the class *Eq* look like? Here's a first attempt:

```
instance Eq (Primitive a) where
  Note d1 x1 == Note d2 x2 = (d1 == d2) ∧ (x1 == x2)
  Rest d1 == Rest d2 = d1 == d2
  _ == _ = False
```

Note the use of (==) on the right-hand side, in several places. Two of those places involve *Dur*, which a type synonym for *Rational*. The *Rational* type is in fact a pre-defined instance of *Eq*, so all is well there. (If it were not an instance of *Eq*, a type error would result.)

But what about the term *x1 == x2*? *x1* and *x2* are values of the polymorphic type *a*, but how do we know that equality is defined on *a*, i.e. that the type *a* is an instance of *Eq*? In fact we don't. The simple fix is to add a constraint to the instance declaration, as follows:

```
instance Eq a ⇒ Eq (Primitive a) where
  Note d1 x1 == Note d2 x2 = (d1 == d2) ∧ (x1 == x2)
  Rest d1 == Rest d2 = d1 == d2
  _ == _ = False
```

This can be read, "For any type *a* in the class *Eq*, the type *Primitive a* is also in the class *Eq*, and here is the definition of (==) for that type." Indeed, it we had written the original type declaration like this:

```
data Primitive a = Note Dur a
                 | Rest Dur
     deriving Eq
```

then Haskell would have derived the above correct instance declaration for us automatically.

So, for example, (==) is defined on the type *Primitive Pitch*, because *Pitch* is a type synonym for (*PitchClass*, *Octave*), and (a) *PitchClass* is an instance of *Eq* by our effort above, (b) *Octave* is a synonym for *Int*, which is an instance of *Eq*, and (c) we mentioned earlier that the pair type is an instance of *Eq*. Indeed, now that we have seen an instance for a polymorphic type, we can understand what the pre-defined instance for polymorphic pairs must look like, namely:

```
instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (x1, y1) == (x2, y2) = (x1 == x2) ∧ (y1 == y2)
```

About the only thing we haven't considered is a *recursive* type. So let's look at *Music*, also from Chapter 2:

```
data Music a = Primitive (Primitive a)
             | Music a :+: Music a
             | Music a :=: Music a
             | Modify Control (Music a)
```

Its instance declaration for *Eq* seems obvious:

```
instance Eq a ⇒ Eq (Music a) where
  Primitive p1 == Primitive p2 = p1 == p2
  (ma1 :+: mb1) == (ma2 :+: mb2) = (ma1 == ma2) ∧ (mb1 == mb2)
  (ma1 :=: mb1) == (ma2 :=: mb2) = (ma1 == ma2) ∧ (mb1 == mb2)
  Modify c1 m1 == Modify c2 m2 = (c1 == c2) ∧ (m1 == m2)
```

Indeed, assuming that we also declare *Control* to be an instance of *Eq*, this is just what we want, and can be automatically derived by adding a **deriving** clause to the data type declaration for *Music*.

In reality, the class *Eq* as defined in Haskell's Standard Prelude is slightly richer than what we defined above. Here is its exact form:

```
class Eq a where
  (==), (≠) :: a → a → Bool
  x ≠ y = ¬ (x == y)
  x == y = ¬ (x ≠ y)
```

This is an example of a class with two operations, one for equality, the other for inequality. It also demonstrates the use of a *default method*, one for each operator. If a method for a particular operation is omitted in an instance declaration, then the default one defined in the class declaration, if it exists, is used instead. For example, all of the instances of *Eq* defined earlier will work perfectly well with the above class declaration, yielding just the right definition of inequality that we want: the logical negation of equality.

> **Details:** Both the inequality and the logical negation operators are shown here using the mathematical notation, $\neq$ and $\neg$, respectively. When writing your Haskell programs, you will have to use the operator /= and the name "*not*," respectively.

11.3 Inheritance

Haskell also supports a notion called *inheritance*. For example, we may wish to define a class *Ord* which "inherits" all of the operations in *Eq*, but in addition has a set of comparison operations and minimum and maximum functions (a fuller definition of *Ord*, as taken from the Standard Prelude, is given in Chapter B):

```
class Eq a ⇒ Ord a where
  (<), (⩽), (⩾), (>) :: a → a → Bool
  max, min :: a → a → a
```

Note the constraint $Eq\ a \Rightarrow$ in the **class** declaration. We say that *Eq* is a *superclass* of *Ord* (conversely, *Ord* is a *subclass* of *Eq*), and any type that is an instance of *Ord* must also be an instance of *Eq*. The reason that this extra constraint makes sense is that to perform comparisons such as $a \leqslant b$ and $a \geqslant b$ implies that we know how to compute $a == b$.

For example, following the strategy we used for *Eq*, we could declare *Music* an instance of *Ord* as follows (note the constraint $Ord\ a \Rightarrow ...$):

```
instance Ord a ⇒ Ord (Music a) where
  Primitive p1 < Primitive p2 = p1 < p2
  (ma1 :+: mb1) < (ma2 :+: mb2) = (ma1 < ma2) ∧ (mb1 < mb2)
  (ma1 :=: mb1) < (ma2 :=: mb2) = (ma1 < ma2) ∧ (mb1 < mb2)
  Modify c1 m1 == Modify c2 m2 = (c1 < c2) ∧ (m1 < m2)
  ...
```

Although this is a perfectly well-defined definition for <, it is not clear that it is the behavior that we want, an issue that we will return to in Section 11.6.

Another benefit of inheritance is shorter constraints. For example, the type of a function that uses operations from both the *Eq* and *Ord* classes can use the constraint $(Ord\ a)$ rather than $(Eq\ a, Ord\ a)$, since *Ord* "implies" *Eq*.

As an example of the use of *Ord*, a generic *sort* function should be able to sort lists of any type that is an instance of *Ord*, and thus its most general type should be:

$$sort :: (Ord\ a) \Rightarrow [\,a\,] \rightarrow [\,a\,]$$

This typing for *sort* would naturally arise through the use of comparison operators such as $<$ and $\geqslant$ in its definition.

> **Details:** Haskell also permits *multiple inheritance*, since classes may have more than one superclass. Name conflicts are avoided by the constraint that a particular operation can be a member of at most one class in any given scope. For example, the declaration
>
> $$\textbf{class}\ (Eq\ a, Show\ a) \Rightarrow C\ a\ \textbf{where}...$$
>
> creates a class C which inherits operations from both *Eq* and *Show*.
>
> Finally, class methods may have additional class constraints on any type variable except the one defining the current class. For example, in this class:
>
> $$\begin{array}{l}\textbf{class}\ C\ a\ \textbf{where}\\ \quad m :: Eq\ b \Rightarrow a \rightarrow b\end{array}$$
>
> the method m requires that type b is in class *Eq*. However, additional class constraints on type a are not allowed in the method m; these would instead have to be part of the constraint in the class declaration.

11.4 Haskell's Standard Type Classes

The Standard Prelude defines many useful type classes, including *Eq* and *Ord*. They are described in detail in Chapter B. In addition, the Haskell Report and the Library Report contain useful examples and discussions of type classes; you should feel encouraged to read through them.

The *Num* class, which we have been using implicitly throughout much of the text, is described in more detail below. With this explanation a few more of Haskell's secrets will be revealed.

11.4.1 The *Num* Class

As you know, Haskell provides several kinds of numbers, some of which we have used already: *Int*, *Integer*, *Rational*, and *Float*. These numbers are instances of various type classes arranged in a rather complicated hierarchy. The reason for this is that there are many operations, such as (+), *abs*, and *sin*, that are common amongst some of these number types. For example, we would expect (+) to be defined on every kind of number, whereas *sin* might only be applicable to either single precision (*Float*) or double-precision (*Double*) floating-point numbers.

Control over which numerical operations are allowed and which aren't is the purpose of the numeric type class hierarchy. At the top of the hierarchy, and therefore containing operations that are valid for all numbers, is the class *Num*. It is defined as:

```
class (Eq a, Show a) ⇒ Num a where
  (+), (−), (∗) :: a → a → a
  negate :: a → a
  abs, signum :: a → a
  fromInteger :: Integer → a
```

Note that (/) is *not* an operation in this class. *negate* is the negation function; *abs* is the absolute value function; and *signum* is the sign function, which returns −1 if its argument is negative, 0 if it is 0, and 1 if it is positive. *fromInteger* converts an *Integer* into a value of type *Num a* ⇒ *a*, which is useful for certain coercion tasks.

> **Details:** Haskell also has a negation operator, which is Haskell's only prefix operator. However, it is just shorthand for *negate*. That is, $-e$ in Haskell is shorthand for *negate e*.
>
> The operation *fromInteger* also has a special purpose. You might have wondered how it is that we can write the number 42, say, both in a context requiring an *Int* and in one requiring a *Float* (say). Somehow Haskell "knows" that the 42 is the one that is required in a given context. But, what is the type of 42 itself? The answer is that it has type *Num a* ⇒ *a*, for some *a* to be determined by its context. (If this seems strange, remember that [ ] by itself is also somewhat ambiguous; it is a list, but a list of what? The best we can say about its type is that it is [*a*] for some *a* yet to be determined.)
>
> The way this is achieved in Haskell is that 42 is actually shorthand for *fromInteger* 42. Since *fromInteger* has type *Num a* ⇒ *Integer* → *a*, then *fromInteger* 42 has type *Num a* ⇒ *a*.

The complete hierarchy of numeric classes is shown in Figure 11.1; note that some of the classes are subclasses of certain non-numeric classes, such as *Eq* and *Show*. The comments below each class name refer to the Standard Prelude types that are instances of that class. See Chapter B for more detail.

The Standard Prelude actually defines only the most basic numeric types: *Int*, *Integer*, *Float* and *Double*. Other numeric types such as rational numbers (*Ratio a*) and complex numbers (*Complex a*) are defined in libraries. The connection between these types and the numeric classes is given in Figure 11.2. The instance declarations implied by this table can be found in the Haskell Report.

11.4.2 The *Show* Class

It is very common to want to convert a data type value into a string. In fact, it happens all the time when we interact with GHCi at the command prompt, and GHCi will complain if it does not "know" how to "show" a value. The type of anything that GHCi prints must be an instance of the *Show* class.

We will not detail all of the methods in the *Show* class, in fact we will only discuss one of them:

```
class Show a where
  show :: a → String
```

Instances of *Show* can be derived, so we normally don't have to worry about the details of the definition of *show*. For example, the actual definition of the *Primitive* type that we gave in Chapter 2 is:

```
data Prim = Note Dur Pitch
          | Rest Dur
    deriving (Show, Eq, Ord)
```

Details: When instances of more than one class are derived for the same data type, they appear grouped in parentheses as above. In this case *Eq* *must* appear if *Ord* does (unless an explicit instance for *Eq* is given), since *Eq* is a superclass of *Ord*.

Lists also have a *Show* instance, but it is not derived, since, after all, lists have special syntax. Also, when *show* is applied to a string such as `"Hello"`, it should generate a string that, when printed, will look like `"Hello"`. This means that it must include characters for the quotation marks themselves, which in Haskell is achieved by prefixing the quotation mark with the "escape" character \. Given the following data declaration:

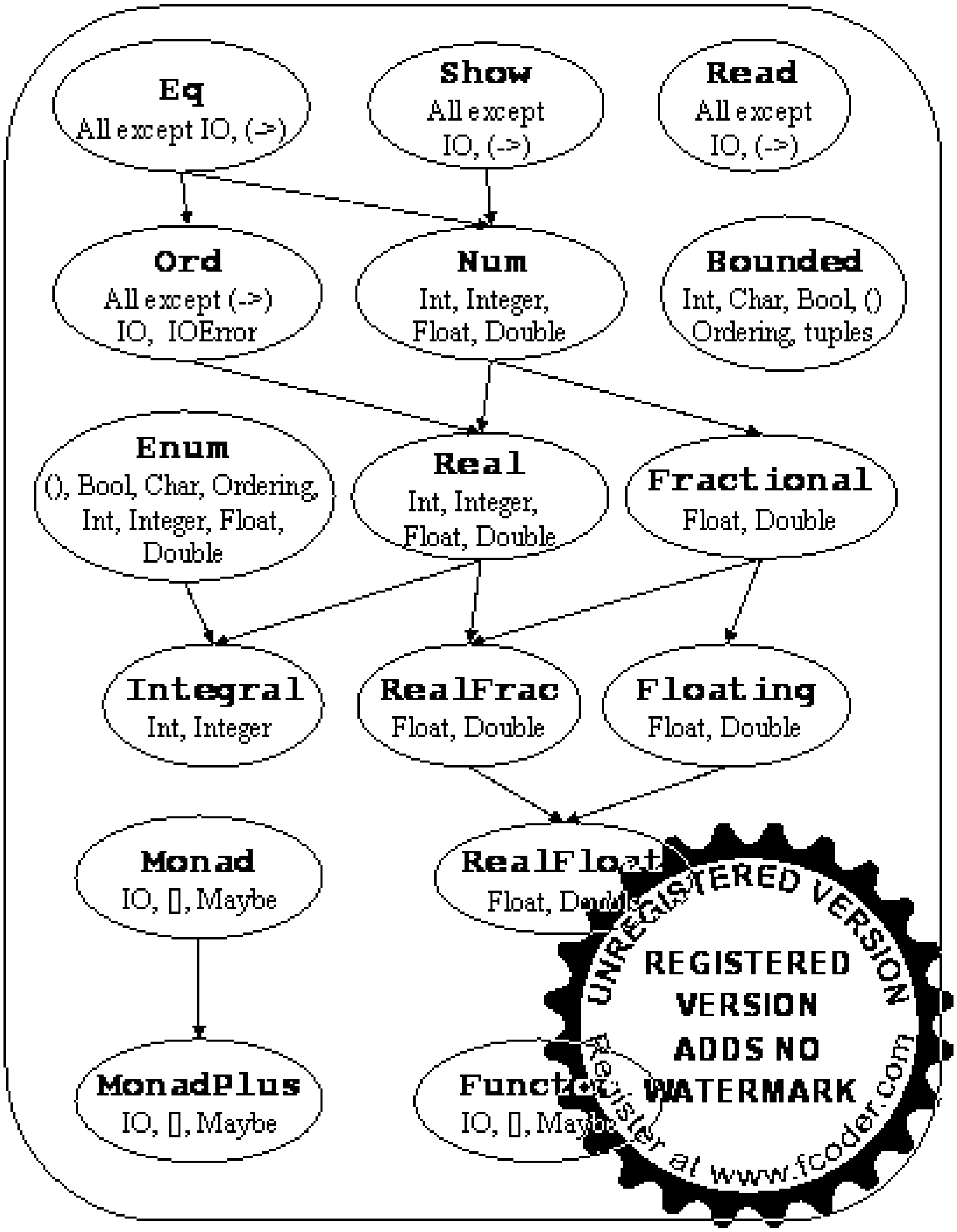


Figure 11.1: Numeric Class Hierarchy

Numeric Type	**Type Class**	**Description**
Int	*Integral*	Fixed-precision integers
Integer	*Integral*	Arbitrary-precision integers
Integral a ⇒ Ratio a	*RealFrac*	Rational numbers
Float	*RealFloat*	Real floating-point, single precision
Double	*RealFloat*	Real floating-point, double precision
RealFloat a ⇒ Complex a	*Floating*	Complex floating-point

Figure 11.2: Standard Numeric Types

```
data Hello = Hello
    deriving Show
```

it is then instructive to ponder over the following calculations:

```
show Hello ⟹ "Hello"
show (show Hello) ⟹ show "Hello" ⟹ "\"Hello\""
show (show (show Hello)) ⟹ "\"\\\"Hello\\\"\""
```

Details: To refer to the escape character itself, it must also be escaped; thus "\\" prints as \.

For further pondering, consider the following program. See if you can figure out what it does, and why![1]

```
main = putStr (quine q)
quine s = s ++ show s
q = "main = putStr (quine q)\nquine s = s ++ show s\nq = "
```

Derived *Show* instances are possible for all types whose component types also have *Show* instances. *Show* instances for most of the standard types are provided in the Standard Prelude.

11.5 Derived Instances

In addition to *Eq* and *Ord*, instances of *Enum*, *Bounded*, *Ix*, *Read*, and *Show* (see Chapter B) can also be generated by the **deriving** clause. These

[1] The essence of this idea is due to Willard Van Orman Quine [Qui66], and its use in a computer program is discussed by Hofstadter [Hof79]. It was adapted to Haskell by Jón Fairbairn.

type classes are widely used in Haskell programming, making the deriving mechanism very useful.

The textual representation defined by a derived *Show* instance is consistent with the appearance of constant Haskell expressions of the type in question. For example, from:

$$\textbf{data}\ \mathit{Color} = \mathit{Red} \mid \mathit{Orange} \mid \mathit{Yellow} \mid \mathit{Green} \mid \mathit{Blue} \mid \mathit{Indigo} \mid \mathit{Violet}$$
$$\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Enum}, \mathit{Show})$$

we can expect that:

$$\mathit{show}\ [\mathit{Red}\mathinner{..}]$$
$$\Longrightarrow \texttt{"[Red,Orange,Yellow,Green,Blue,Indigo,Violet]"}$$

Further details about derived instances can be found in the Haskell Report.

Many of the pre-defined data types in Haskell have **deriving** clauses, even ones with special syntax. For example, if we could write a data type declaration for lists it would look something like this:

$$\begin{aligned}&\textbf{data}\ [a] = [\,]\\ &\qquad\qquad \mid a : [a]\\ &\quad\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord})\end{aligned}$$

The derived *Eq* and *Ord* instances for lists are the usual ones; in particular, character strings, as lists of characters, are ordered as determined by the underlying *Char* type, with an initial sub-string being less than a longer string; for example, `"cat"` < `"catalog"` is *True*.

In practice, *Eq* and *Ord* instances are almost always derived, rather than user-defined. In fact, you should provide your own definitions of equality and ordering predicates only with some trepidation, being careful to maintain the expected algebraic properties of equivalence relations and total orders (more on this later). An intransitive (==) predicate, for example, could be disastrous, confusing readers of the program who may expect (==) to be transitive. Nevertheless, it is sometimes necessary to provide *Eq* or *Ord* instances different from those that would be derived.

11.6 Reasoning With Type Classes

Type classes often imply a set of *laws* which govern the use of the operators in the class. For example, for the *Eq* class, we can expect the following laws to apply for every instance of the class:

$$\begin{array}{l}
(x \neq y) = \neg\,(x == y) \\
(x == y) \wedge (y == z) \supseteq (x == z)
\end{array}$$

where $\supseteq$ should be read "implies that."

However, there is no way to guarantee these laws. A user may create an instance of *Eq* that violates them, and in general Haskell has no way to enforce them. Nevertheless, it is useful to state the laws that interest us for a certain class, and to state the expectation that all instances of the class be "law-abiding." Then as diligent functional programmers, we should ensure that every instance that we write, whether for our own class or someone else's, is in fact law-abiding.

As another example, consider the *Ord* class, whose instances are intended to be *totally ordered*, which means that the following laws should hold, for all a, b, and c:

$$\begin{array}{l}
a \leqslant a \\
(a \leqslant b) \wedge (b \leqslant c) \supseteq (a \leqslant c) \\
(a \leqslant b) \wedge (b \leqslant a) \supseteq (a == b) \\
(a \neq b) \supseteq (a < b) \vee (b < a)
\end{array}$$

Similar laws should hold for $(>)$.

But alas, our instance of *Music* in the class *Ord* given in Section 11.3 does not satisfy all of these laws! To see why, suppose we have two *Primitive* values $p1$ and $p2$ such that $p1 < p2$. Now consider these two *Music* values:

```
m1 = Primitive p1 :+: Primitive p2
m2 = Primitive p2 :+: Primitive p1
```

Clearly $m1 == m2$ is false, but the problem is, so are $m1 < m2$ and $m2 < m1$, thus violating the last law above.

To fix the problem, we need to use a *lexicographic ordering* on the *Music* type, such as used in a dictionary. For example, "polygon" comes before "polymorphic," using a left-to-right comparison of the letters. The new instance declaration looks like this:

```
instance Ord a ⇒ Ord (Music a) where
  Primitive p1 < Primitive p2 = p1 < p2
  Primitive p1 < _ = True
  (ma1 :+: mb1) < Primitive = False
  (ma1 :+: mb1) < (ma2 :+: mb2) = (ma1 < ma2) ∨ (ma1 == ma2) ∧ (mb1 < mb2)
  (ma1 :+: mb1) < _ = True
  (ma1 :=: mb1) < Primitive = False
```

$$\begin{array}{l}
(\mathit{ma1} :\!=\!: \mathit{mb1}) < (\mathit{ma2} :\!+\!: \mathit{mb2}) = \mathit{False} \\
(\mathit{ma1} :\!=\!: \mathit{mb1}) < (\mathit{ma2} :\!=\!: \mathit{mb2}) = (\mathit{ma1} < \mathit{ma2}) \lor (\mathit{ma1} == \mathit{ma2}) \land (\mathit{mb1} < \mathit{mb2}) \\
(\mathit{ma1} :\!=\!: \mathit{mb1}) < \_ = \mathit{True} \\
\mathit{Modify}\ \mathit{c1}\ \mathit{m1} < \mathit{Modify}\ \mathit{c2}\ \mathit{m2} = (\mathit{c1} < \mathit{c2}) \lor (\mathit{c1} == \mathit{c2}) \land (\mathit{m1} < \mathit{m2}) \\
\mathit{Modify}\ \mathit{c1}\ \mathit{m1} < \_ = \mathit{False}
\end{array}$$

This example shows the value of checking to be sure that each of your instances obeys the laws of its class. Of course, that check should come in the way of a proof. This example also highlights the utility of derived instances, since the derived instance of *Music* for the class *Ord* is equivalent to that above, yet is done automatically.

Exercise 11.1 Prove that the instance of *Music* in the class *Eq* satisfies the laws of its class. Also prove that the modified instance of *Music* in the class *Ord* satisfies the laws of its class.

Exercise 11.2 Write out appropriate instance declarations for the *Color* type in the classes *Eq*, *Ord*, and *Enum*.

Chapter 12

Random Numbers, Probability Distributions, and Markov Chains

```
module Haskore.RandomMusic where

import Haskore
import System.Random
import System.Random.Distributions
import qualified Data.MarkovChain as M
```

The use of randomness in composition can be justified by the somewhat random, exploratory nature of the creative mind, and indeed it has been used in computer music composition for many years. In this chapter we will explore several sources of random numbers and how to use them in generating simple melodies. With this foundation you will hopefully be able to use randomness in more sophisticated ways in your compositions. Music relying at least to some degree on randomness is said to be *stochastic*, or *aleatoric*.

12.1 Random Numbers

This section describes the basic functionality of Haskell's *System.Random* module, which is a library for random numbers. The library presents a fairly abstract interface that is structured in two layers of type classes: one that captures the notion of a *random generator*, and one for using a random generator to create *random sequences*.

We can create a random number generator using the built-in *mkStdGen* function:

```
mkStdGen :: Int → StdGen
```

which takes an *Int* seed as argument, and returns a "standard generator" of type *StdGen*. For example, we can define:

```
sGen :: StdGen
sGen = mkStdGen 42
```

We will use this single random generator quite extensively in the remainder of this chapter.

StdGen is an instance of *Show*, and thus its values can be printed—but they appear in a rather strange way, basically as two integers. Try typing *sGen* to the GHCi prompt.

More importantly, *StdGen* is an instance of the *RandomGen* class:

```
class RandomGen g where
  genRange :: g → (Int, Int)
  next :: g → (Int, g)
  split :: g → (g, g)
```

The reason that *Int*s are used here is that essentially all pseudo-random number generator algorithms are based on a fixed-precision binary number, such as *Int*. We will see later how this can be coerced into other number types.

For now, try applying the operators in the above class to the *sGen* value above. The *next* function is particularly important, as it generates the next random number in a sequence as well as a new random number generator, which in turn can be used to generate the next number, and so on. It should be clear that we can then create an infinite list of random *Int*s like this:

```
randInts :: StdGen → [Int]
randInts g = let (x, g′) = next g
             in x : randInts g′
```

Look at the value *take* 10 (*randInts sGen*) to see a sample output.

To support other number types, the *Random* library defines this type class:

```
class Random a where
   randomR :: RandomGen g ⇒ (a, a) → g → (a, g)
   random :: RandomGen g ⇒ g → (a, g)

   randomRs :: RandomGen g ⇒ (a, a) → g → [a]
   randoms :: RandomGen g ⇒ g → [a]

   randomRIO :: (a, a) → IO a
   randomIO :: IO a
```

Built-in instances of *Random* are provided for *Int*, *Integer*, *Float*, *Double*, *Bool*, and *Char*.

The set of operators in the *Random* class is rather daunting, so let's focus on just one of them for now, namely the third one, *RandomRs*, which is also perhaps the most useful one. This function takes a random number generator (such as *sGen*), along with a range of values, and generates an infinite list of random numbers within the given range (the pair representing the range is treated as a closed interval). Here are several examples of this idea:

```
randFloats :: [Float]
randFloats = randomRs (−1, 1) sGen

randIntegers :: [Integer]
randIntegers = randomRs (0, 100) sGen

randString :: String
randString = randomRs ('a', 'z') sGen
```

Recall that a string is a list of characters, so we choose here to use the name *randString* for our infinite list of characters. If you believe the story about a monkey typing a novel, then you might believe that *randString* contains something interesting to read.

So far we have used a seed to initialize our random number generators, and this is good in the sense that it allows us to generate repeatable, and therefore more easily testable, results. If instead you prefer a non-repeatable result, in which you can think of the seed as being the time of day when the program is executed, then you need to use a function that is in the IO monad. The last two operators in the *Random* class server this purpose. For example, consider:

```
randIO :: IO Float
randIO = randomRIO (0, 1)
```

If you repeatedly type *randIO* at the GHCi prompt, it will return a different random number every time. This is clearly not purely "functional," and is why it is in the IO monad. As another example:

```
randIO′ :: IO ()
randIO′ = do r1 ← randomRIO (0, 1) :: IO Float
             r2 ← randomRIO (0, 1) :: IO Float
             print (r1 == r2)
```

will essentially always return *False*. (The type signature is needed to ensure that the value generated has an unambigous type.)

12.2 Probability Distributions

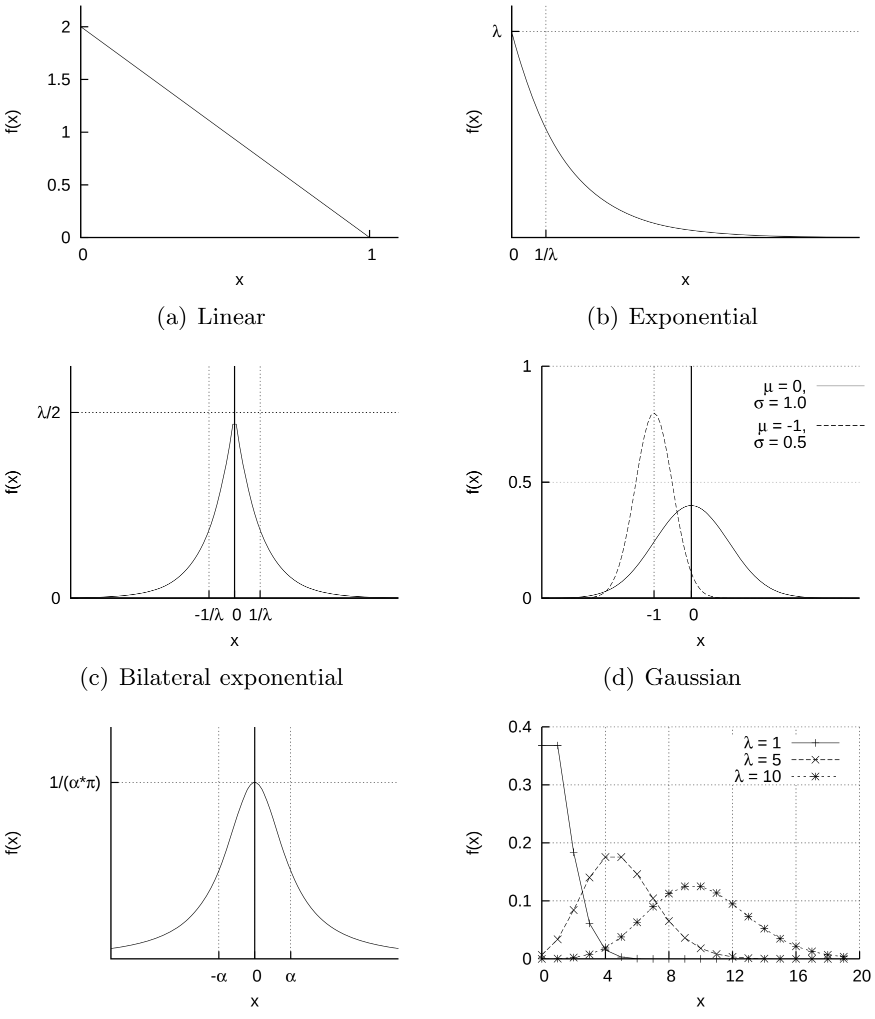
The random number generators described in the previous section are assumed to be *uniform*, meaning that the probability of generating a number within a given interval is the same everywhere in the range of the generator. For example, in the case of *Float* (that purportedly represents *continuous* real numbers), suppose we are generating numbers in the range 0 to 10. Then we would expect the probability of a number appearing in the range 2.3-2.4 to be the same as the probability of a number appearing in the range 7.6-7.7, namely 0.01, or 1% (i.e. 0.1/10). In the case of *Int* (a *discrete* or *integral* number type), we would expect the probability of generating a 5 to be the same as generating an 8. In both cases, we say that we have a *uniform distribution.*

But we don't always want a uniform distribution. In generating music, in fact, it's often the case that we want some kind of a non-uniform distribution. Mathematically, the best way to describe a distribution is by plotting how the probability changes over the range of values that it produces. In the case of continuous numbers, this is called the *probability density function*, which has the property that its integral over the full range of values is equal to 1.

The *System.Random.Distributions* library provides a number of different probability distributions, which are described below. Figure 12.1 shows the probability density functions for each of othem.

Here is a list and brief description of each random number generator:

linear Generates a *linearly* distributed random variable between 0 and 1.



(a) Linear

(b) Exponential

(c) Bilateral exponential

(d) Gaussian

(e) Cauchy

(f) Poisson

Figure 12.1: Various Probability Density Functions

The probability density function is given by:

$$f(x) = \begin{cases} 2(1-x) & \text{if } 0 \leqslant x \leqslant 1 \\ 0 & \text{otherwise} \end{cases}$$

The type signature is:

$$\begin{aligned} &\mathit{linear} :: (\mathit{RandomGen}\ g, \mathit{Floating}\ a, \mathit{Random}\ a, \mathit{Ord}\ a) \Rightarrow \\ &\qquad g \to (a, g) \end{aligned}$$

The mean value of the linear distribution is 1/3.

exponential Generates an *exponentially* distributed random variable given a spread parameter λ. A larger spread increases the probability of generating a small number. The mean of the distribution is $1/\lambda$. The range of the generated number is conceptually 0 to ∞, although the chance of getting a very large number is very small. The probability density function is given by:

$$f(x) = \lambda e^{-\lambda x}$$

The type signature is:

$$\begin{aligned} &\mathit{exponential} :: (\mathit{RandomGen}\ g, \mathit{Floating}\ a, \mathit{Random}\ a) \Rightarrow \\ &\qquad\qquad a \to g \to (a, g) \end{aligned}$$

The first argument is the parameter λ.

bilateral exponential Generates a random number with a *bilateral exponential* distribution. Similar to exponential, but the mean of the distribution is 0 and 50% of the results fall between $-1/\lambda$ and $1/\lambda$. The probability density function is given by:

$$f(x) = \frac{1}{2}\lambda e^{-\lambda|x|}$$

The type signature is:

$$\begin{aligned} &\mathit{bilExp} :: (\mathit{Floating}\ a, \mathit{Ord}\ a, \mathit{Random}\ a, \mathit{RandomGen}\ g) \Rightarrow \\ &\qquad a \to g \to (a, g) \end{aligned}$$

Gaussian Generates a random number with a *Gaussian*, also called *normal*, distribution, given mathematically by:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where σ is the *standard deviation*, and μ is the *mean*. The type signature is:

$$\begin{aligned} &\mathit{gaussian} :: (\mathit{Floating}\ a, \mathit{Random}\ a, \mathit{RandomGen}\ g) \Rightarrow \\ &\qquad\qquad a \rightarrow a \rightarrow g \rightarrow (a, g) \end{aligned}$$

The first argument is the standard deviation σ and the second is the mean μ. Probabilistically, about 68.27% of the numbers in a Gaussian distribution fall within $\pm\sigma$ of the mean; about 95.45% are within $\pm 2\sigma$, and 99.73% are within $\pm 3\sigma$.

Cauchy Generates a *Cauchy*-distributed random variable. The distribution is symmetric with a mean of 0. The density function is given by:

$$f(x) = \frac{\alpha}{\pi(\alpha^2 + x^2)}$$

As with the Gaussian distribution, it is unbounded both above and below the mean, but at its extremes it approaches 0 more slowly than the Gaussian. The type signature is:

$$\begin{aligned} &\mathit{cauchy} :: (\mathit{Floating}\ a, \mathit{Random}\ a, \mathit{RandomGen}\ g) \Rightarrow \\ &\qquad\qquad a \rightarrow g \rightarrow (a, g) \end{aligned}$$

The first argument corresponds to α above, and is called the *density*.

Poisson Generates a *Poisson*-distributed random variable. The Poisson distribution is discrete, and generates only non-negative numbers. λ is the mean of the distribution. If λ is an integer, the probability that the result is $j = \lambda - 1$ is the same as that of $j = \lambda$. The probability of generating the number j is given by:

$$P\{X = j\} = \frac{\lambda^j}{j!} e^{-\lambda}$$

The type signature is:

```
poisson :: (Num t, Ord a, Floating a, RandomGen g, Random a) ⇒
             a → g → (t, g)
```

Custom Sometimes it is useful to define one's own discrete probability distribution function, and to generate random numbers based on it. The function *frequency* does this—given a list of weight-value pairs, it generates a value randomly picked from the list, weighting the probability of choosing each value by the given weight.

```
frequency :: (Floating w, Ord w, Random w, RandomGen g) ⇒
               [(w, a)] → g → (a, g)
```

12.2.1 Random Melodies and Random Walks

Note that each of the non-uniform distribution random number generators described in the last section takes zero or more parameters as arguments, along with a uniform random number generator, and returns a pair consisting of the next random number and a new generator. In other words, the tail end of each type signature has the form:

```
... → g → (a, g)
```

where g is the type of the random number generator, and a is the type of the next value generated.

Given such a function, we can generate an infinite sequence of random numbers with the given distribution in a way similar to what we did earlier for *randInts*. In fact the following function is defined in the *Distributions* library to make this easy:

```
rands :: (RandomGen g, Random a) ⇒ (g → (a, g)) → g → [a]
rands f g = x : rands f g' where (x, g') = f g
```

Let's work through a few musical examples. One thing we will need to do is convert a floating point number to an absolute pitch:

```
toAbsP1 :: Float → AbsPitch
toAbsP1 x = round (40 ∗ x + 30)
```

This function converts a number in the range 0 to 1 into an absolute pitch in the range 30 to 70.

And as we have often done, we will also need to convert an absolute pitch into a note, and a sequence of absolute pitches into a melody:

```
mkNote1 :: AbsPitch → Music Pitch
mkNote1 = note tn ∘ pitch

mkLine1 :: [AbsPitch] → Music Pitch
mkLine1 rands = line (take 32 (map mkNote1 rands))
```

With these functions in hand, we can now generate sequences of random numbers with a variety of distributions, and convert each of them into a melody. For example:

```
    -- uniform distribution
m1 :: Music Pitch
m1 = mkLine1 (randomRs (30, 70) sGen)

    -- linear distribution
m2 :: Music Pitch
m2 = let rs1 = rands linear sGen
     in mkLine1 (map toAbsP1 rs1)

    -- exponential distribution
m3 :: Float → Music Pitch
m3 lam = let rs1 = rands (exponential lam) sGen
         in mkLine1 (map toAbsP1 rs1)

    -- Gaussian distribution
m4 :: Float → Float → Music Pitch
m4 sig mu = let rs1 = rands (gaussian sig mu) sGen
            in mkLine1 (map toAbsP1 rs1)
```

Exercise 12.1 Try playing each of the above melodies, and listen to the musical differences. For *lam*, try values of 0.1, 1, 5, and 10. For *mu*, a value of 0.5 will put the melody in the central part of the scale range—then try values of 0.01, 0.05, and 0.1 for *sig*.

Exercise 12.2 Do the following:

- Try using some of the other probability distributions to generate a melody.

- Instead of using a chromatic scale, try using a diatonic or pentatonic scale.

- Try using randomness to control parameters other than pitch—in particular, duration and/or volume.

Another approach to generating a melody is sometimes called a *random walk*. The idea is to start on a particular note, and treat the sequence of random numbers as *intervals*, rather than as pitches. To prevent the melody from wandering too far from the starting pitch, one should use a probability distribution whose mean is zero. This comes for free with something like the bilateral exponential, and is easily obtained with a distribution that takes the mean as a parameter (such as the Gaussian), but is also easily achieved for other distributions by simply subtracting the mean. To see these two situations, here are random melodic walks using first a Gaussian and then an exponential distribution:

```
      -- Gaussian distribution with mean set to 0
m5 :: Float → Music Pitch
m5 sig = let rs1 = rands (gaussian sig 0) sGen
         in mkLine2 50 (map toAbsP2 rs1)

      -- exponential distribution with mean adjusted to 0
m6 :: Float → Music Pitch
m6 lam = let rs1 = rands (exponential lam) sGen
         in mkLine2 50 (map (toAbsP2 ∘ subtract (1 / lam)) rs1)

toAbsP2 :: Float → AbsPitch
toAbsP2 x = round (5 ∗ x)

mkLine2 :: AbsPitch → [AbsPitch] → Music Pitch
mkLine2 start rands = line (take 64 (map mkNote1 (scanl (+) start rands)))
```

Note that *toAbsP2* does something reasonable to interpret a floating-point number as an interval, and *mkLine2* uses *scanl* to generate a "running sum" that represents the melody line.

12.3 Markov Chains

Each number in the random number sequences that we have described thus far is *independent* of any previous values in the sequence. This is like flipping a coin—each flip has a 50% chance of being heads or tails, i.e. it is independent of any previous flips, even if the last ten flips were all heads.

	C	*D*	*E*	*F*
C	0.4	0.2	0.2	0.2
D	0.3	0.2	0.0	0.5
E	0.1	0.6	0.1	0.2
F	0.2	0.3	0.3	0.2

Table 12.1: Second-Order Markov Chain

Sometimes, however, we would like the probability of a new choice to depend upon some number of previous choices. This is called a *conditional probability.* In a discrete system, if we look only at the previous value to help determine the next value, then these conditional probabilities can be conveniently represented in a matrix. For example, if we are choosing between the pitches C, D, E, and F, then Table 12.1 might represent the conditional probabilities of each possible outcome. The previous pitch is found in the left column—thus note that the sum of each row is 1.0. So, for example, the probability of choosing a D given that the previous pitch was an E is 0.6, and the probability of an F occurring twice in succession is 0.2.

This idea can of course be generalized to arbitrary numbers of previous events, and in general an $(n + 1)$-dimensional array can be used to store the various conditional probabilities. The number of previous values observed is called the *order* of the Markov Chain.

[TO DO: write the Haskell code to implement this]

12.3.1 Training Data

Instead of generating the conditional probability table ourselves, another approach is to use *training data* from which the conditional probabilities can be *inferred.* This is handy for music, because it means that we can feed in a bunch of melodies that we like, including melodies written by the masters, and use that as a stochastic basis for generating new melodies.

[TO DO: Give some pointers to the literatue, in particular David Cope's work.]

The *Data.MarkovChain* library provides this functionality through a function called *run*, whose type signature is:

```
run :: (Ord a, RandomGen g) ⇒
          Int     -- order of Markov Chain
      → [a]       -- training sequence (treated as circular list)
      → Int       -- index to start within the training sequence
      → g      -- random number generator
      → [a]
```

The *runMulti* function is similar, except that it takes a list of training sequences as input, and returns a list of lists as its result, each being an independent random walk whose probabilities are based on the training data. The following examples demonstrate how to use these functions.

```
      -- some sample training sequences
ps0, ps1, ps2 :: [Pitch]
ps0 = [(C, 4), (D, 4), (E, 4)]
ps1 = [(C, 4), (D, 4), (E, 4), (F, 4), (G, 4), (A, 4), (B, 4)]
ps2 = [(C, 4), (E, 4), (G, 4), (E, 4), (F, 4), (A, 4), (G, 4), (E, 4),
       (C, 4), (E, 4), (G, 4), (E, 4), (F, 4), (D, 4), (C, 4)]

      -- functions to package up run and runMulti
mc ps n = mkLine3 (M.run n ps 0 (mkStdGen 42))
mcm pss n = mkLine3 (concat (M.runMulti n pss 0 (mkStdGen 42)))

      -- music-making functions
mkNote3 :: Pitch → Music Pitch
mkNote3 = note tn

mkLine3 :: [Pitch] → Music Pitch
mkLine3 ps = line (take 64 (map mkNote3 ps))
```

Here are some things to try with the above definitions:

- *mc ps0* 0 will generate a completely random sequence, since it is a "zeroth-order" Markov Chain that does not look at any previous output.
- *mc ps0* 1 looks back one value, which is enough in the case of this simple training sequence to generate an endless sequence of notes that sounds just like the training data. Using any order higher than 1 generates the same result.
- *mc ps1* 1 also generates a result that sounds just like its training data.
- *mc ps2* 1, on the other hand, has some (random) variety to it, because the training data has more than one occurrence of most of the notes. If we increase the order, however, the output will sound more and more like the training data.

- *mcm* [*ps0*, *ps2*] 1 and *mcm* [*ps1*, *ps2*] 1 generate perhaps the most interesting results yet, in which you can hear aspects of both the ascending melodic nature of *ps0* and *ps1*, and the harmonic structure of *ps2*.

- *mcm* [*ps1*, *reverse ps1*] 1 has, not suprisingly, both ascending and descending lines in it, as reflected in the training data.

Exercise 12.3 Play with Markov Chains. Use them to generate more melodies, or to control other aspects of the music, such as rhythm. Also consider other kinds of training data rather than simply sequences of pitches.

Chapter 13

From Performance to Midi

```
module Haskore.ToMidi (toMidi, UserPatchMap, defST)
         where
import Haskore.Music
import Haskore.Performance
import Haskore.GeneralMidi
import Data.List (partition)
import Data.Char (toLower, toUpper)
import Codec.Midi
```

Midi is shorthand for "Musical Instrument Digital Interface," and is a standard protocol for controlling electronic musical instruments. This chapter describes how to convert an abstract *performance* as defined in Chapter 6 into a *standard Midi file* that can be played on any modern PC with a standard sound card.

13.1 An Introduction to Midi

Midi is a standard adopted by most, if not all, manufacturers of electronic instruments and personal computers. At its core is a protocol for communicating *musical events* (note on, note off, etc.) and so-called *meta events* (select synthesizer patch, change tempo, etc.). Beyond the logical protocol, the Midi standard also specifies electrical signal characteristics and cabling details, as well as a *standard Midi file* which any Midi-compatible software package should be able to recognize.

Most "sound-blaster"-like sound cards on conventional PC's know about Midi. However, the sound generated by such modules, and the sound produced from the typically-scrawny speakers on most PC's, is often quite poor.

Family	Program #		Family	Program #
Piano	1-8		Reed	65-72
Chromatic Percussion	9-16		Pipe	73-80
Organ	17-24		Synth Lead	81-88
Guitar	25-32		Synth Pad	89-96
Bass	33-40		Synth Effects	97-104
Strings	41-48		Ethnic	105-112
Ensemble	49-56		Percussive	113-120
Brass	57-64		Sound Effects	121-128

Table 13.1: General Midi Instrument Families

It is best to use an outboard keyboard or tone generator, which are attached to a computer via a Midi interface and cables. It is possible to connect several Midi instruments to the same computer, with each assigned to a different *channel*. Modern keyboards and tone generators are quite good. Not only is the sound excellent (when played on a good stereo system), but they are also *multi-timbral*, which means they are able to generate many different sounds simultaneously, as well as *polyphonic*, meaning that simultaneous instantiations of the same sound are possible.

13.1.1 General Midi

Over the years musicians and manufacturers decided that they also wanted a standard way to refer to commonly used instrument sounds, such as "acoustic grand piano," "electric piano," "violin," and "acoustic bass," as well as more exotic sounds such as "chorus aahs," "voice oohs," "bird tweet," and "helicopter." A simple standard known as *General Midi* was developed to fill this role. The General Midi standard establishes standard names for 128 common instrument sounds (also called "patches") and assigns an integer called the *program number* (also called "program change number"), to each of them. The instrument names and their program numbers are grouped into "familes" of instrument sounds, as shown in Table 13.1.

Now recall that in Chapter 2 we defined a set of instruments via the *InstrumentName* data type (see Figure 2.1). All of the names chosen for that data type come directly from the General Midi standard, except for two, *Percussion* and *Custom*, which were added for convenience and extensibility. By listing the constructors in the order that reflects this assignment, we can derive an *Enum* instance for *InstrumentName* that defines the method *toEnum* that essentially does the conversion from instrument name to program number for us. We can then define a function:

```
toGM :: InstrumentName → ProgNum
toGM Percussion = 0
toGM (Custom name) = 0
toGM in = fromEnum in

type ProgNum = Int
```

that takes care of the two extra cases, which are simply assigned to program number 0.

The derived *Enum* instance also defines a function *fromEnum* that converts program numbers to instrument names. We can then define:

```
fromGM :: ProgNum → InstrumentName
fromGM pn | pn ⩾ 0 ∧ pn ⩽ 127 = fromEnum pn
fromGM pn = error ("fromGM: " ⧺ show pn⧺
                   " is not a valid General Midi program number")
```

> **Details:** Design bug: Because the *IntrumentName* data type contains a non-nullary constructor, namely *Custom*, the *Enum* instance cannot be derived. For now it is defined in the module *GeneralMidi*, but a better solution is to redefine *InstrumentName* in such a way as to avoid this.

13.1.2 Channels and Patch Maps

A Midi *channel* is in essence a programmable instrument. You can have up to 16 channels, numbered 0 through 15, each assigned a different program number (corresponding to an instrument sound, see above). All of the dynamic "Note On" and "Note Off" messages (to be defined shortly) are tagged with a channel number, so up to 16 different instruments can be controlled independently and simultaneously.

The assignment of Midi channels to instrument names is called a *patch map*, and we define a simple association list to capture its structure:

```
type UserPatchMap = [(InstrumentName, Channel)]

type Channel = Int
```

The only thing odd about Midi Channels is that General Midi specifies that Channel 10 (9 in Haskore's 0-based numbering) is dedicated to *percussion* (which is different from the "percussive instruments" described in Table 13.1). When Channel 10 is used, any program number to which it is assigned is ignored, and instead each note corresponds to a different percussion sound. In particular, General Midi specifies that the notes corresponding to Midi Keys 35 through 82 correspond to specific percussive sounds. Indeed, recall that in Chapter 5 we in fact captured these percussion sounds through the *PercussionSound* data type, and we defined a way to convert such a sound into an absolute pitch (i.e. *AbsPitch*). Haskore's absolute pitches, by the way, are in one-to-one correspondence with Midi Key nunmbers.

Except for percussion, the Midi Channel used to represent a particular instrument is completely arbitrary. Indeed, it is tedious to explicitly define a new patch map every time the instrumentation of a piece of music is changed. Therefore it is convenient to define a function that automatically creates a *UserPatchMap* from a list of instrument names:

```
makeGMMap :: [InstrumentName] → UserPatchMap
makeGMMap ins = mkGMMap 0 ins
  where mkGMMap _ [] = []
        mkGMMap n _ | n ⩾ 15 =
                      error "Too many instruments; not enough Midi channels."
        mkGMMap n (Percussion : ins) =
                      (Percussion, 9) : mkGMMap n ins
        mkGMMap n (i : ins) =
                      (i, chanList !! n) : mkGMMap (n + 1) ins
        chanList = [0 .. 8] ++ [10 .. 15]      -- channel 9 is for percussion
```

Note that, since there are only 15 Midi channels plus percussion, we can handle only 15 different instruments, and an error is signaled if this limit is exceeded.[1]

Finally, we define a function to look up an *InstrumentName* in a *UserPatchMap*, and return the associated channel as well as its program number:

```
upmLookup :: UserPatchMap → InstrumentName → (Channel, ProgNum)
upmLookup upm iName = (channel, toGM iName)
  where channel = maybe (error ("instrument " ++ show iName ++
                                 " not in patch map"))
                        id (lookup iName upm)
```

[1] It is conceivable to define a function to test whether or not two tracks can be combined with a Program Change (tracks can be combined if they don't overlap), but this remains for future work.

13.1.3 Standard Midi Files

The Midi standard defines the precise format of a *standard Midi file.* At the time when the Midi standard was first created, disk space was at a premium, and thus a compact file structure was important. Standard Midi files are thus defined at the bit and byte level, and are quite compact. We are not interested in this low-level representation (any more than we are interested in the signals that run on Midi cables), and thus in Haskore we take a more abstract approach: We define an algebraic data type called *Midi* to capture the abstract structure of a standard Midi file, and then define functions to convert values of this data type to and from actual Midi files. This separation of concerns makes the structure of the Midi file clearer, makes debugging easier, and provides a natural path for extending Haskore's functionality with direct Midi capability (discussed further in Chapter **??**).

We will not discuss the details of the functions that read and write the actual Midi files; the interested reader may find them in the modules *ReadMidi* and *OutputMidi*, respectively. Instead, we will focus on the *Midi* data type, which is defined in the module *Codec.Midi*. We do not need all of its functionality, and thus we show in Figure **??** only those parts of the module that we need for this chapter. Here are the salient points about this data type and the structure of Midi files:

1. There are three types of Midi files:

    - A Format 0, or *SingleTrack*, Midi file stores its information in a single track of events, and is best used only for monophonic music.
    - A Format 1, or *MultiTrack*, Midi file stores its information in multiple tracks that are played simultaneously, where each track normally corresponds to a single Midi Channel.
    - A Format 2, or *MultiPattern*, Midi file also has multiple tracks, but they are temporally independent.

    In this chapter we only use *SingleTrack* and *MultiTrack* Midi files, depending on how many Channels we need.

2. The *TimeDiv* field refers to the *time-code division* used by the Midi file. We will always use 96 time divisions, or "ticks," per quarternote, and thus this field will always be *TicksPerBeat* 96.

3. The main body of a Midi file is a list of *Track*s, each of which in turn is a list of time-stamped (in number of ticks) *Message*s (or "events").

```
    -- From the Codec.Midi module

data Midi = Midi{fileType :: FileType,
                 timeDiv :: TimeDiv
                 tracks :: [Track Ticks]}
     deriving (Eq, Show)

data FileType = SingleTrack | MultiTrack | MultiPattern
     deriving (Eq, Show)

type Track a = [(a, Message)]

data TimeDiv = TicksPerBeat Int      -- 1 through (2^15 - 1)
             | ...
     deriving (Show, Eq)

type Ticks = Int      -- 0 through (2^28 - 1)
type Time = Double
type Channel = Int      -- 0 through 15
type Key = Int      -- 0 through 127
type Velocity = Int      -- 0 through 127
type Pressure = Int      -- 0 through 127
type Preset = Int      -- 0 through 127
type Tempo = Int      -- microseconds per beat, 1 through (2^24 - 1)

data Message =
         -- Channel Messages
       NoteOff{channel :: !Channel, key :: !Key, velocity :: !Velocity}
     | NoteOn{channel :: !Channel, key :: !Key, velocity :: !Velocity}
     | ProgramChange{channel :: !Channel, preset :: !Preset}
     | ...
         -- Meta Messages
     | TempoChange ! Tempo |
     | ...
  deriving (Show, Eq)

fromAbsTime :: (Num a) ⇒ Track a → Track a
fromAbsTime trk = zip ts′ ms
  where (ts, ms) = unzip trk
        (_, ts′) = mapAccumL (λacc t → (t, t − acc)) 0 ts
```

Figure 13.1: Partial Definition of the *Midi* Data Type

4. There are two kinds of *Messages*: *channel messages* and *meta messages*. Figure 13.1 shows just those messages that we are interested in:

   (a) *NoteOn ch k v* turns on key (pitch) *k* with velocity (volume) *v* on Midi channel *ch*. The velocity is an integer in the range 0 to 127.

   (b) *NoteOff ch k v* performs a similar function in turning the note off.

   (c) *ProgChange ch pr* sets the program number for channel *ch* to *pr*. This is how an instrument is selected.

   (d) *TempoChange t* sets the tempo to *t*, which is the time, in microseconds, of one whole note. Using 120 beats per minute as the norm, or 2 beats per second, that works out to 500,000 microseconds per beat, which is the default value that we will use.

13.2 Converting a Performance into Midi

Our goal is to convert a value of type *Performance* into a value of type *Midi*. We can summarize the situation pictorially as follows ...

Given a *UserPatchMap*, a *Performance* is converted into a *Midi* value by the *toMidi* function. If the given *UserPatchMap* is invalid, it creates a new one using *makeGMMap* described earlier.

```
toMidi :: Performance → UserPatchMap → Midi
toMidi pf upm =
  let splitList = splitByInst pf
      insts = map fst splitList
      rightMap = if (allValid upm insts) then upm
                 else (makeGMMap insts)
  in Midi (if length splitList == 1 then SingleTrack else MultiTrack)
          (TicksPerBeat division)
          (map (fromAbsTime ∘ performToMEvs rightMap) splitList)

division = 96 :: Int
```

The following function is used to test whether or not every instrument in a list is found in a *UserPatchMap*:

```
allValid :: UserPatchMap → [InstrumentName] → Bool
allValid upm = and ∘ map (lookupB upm)
```

```
lookupB :: UserPatchMap → InstrumentName → Bool
lookupB upm x = or (map ((== x) ∘ fst) upm)
```

The strategy is to associate each channel with a separate track. Thus we first partition the event list into separate lists for each instrument, and signal an error if there are more than 16:

```
splitByInst :: Performance → [(InstrumentName, Performance)]
splitByInst [] = []
splitByInst pf = (i, pf1) : splitByInst pf2
        where i = eInst (head pf)
              (pf1, pf2) = partition (λe → eInst e == i) pf
```

Note how *partition* is used to group into *pf1* those events that use the same instrument as the first event in the performance. The rest of the events are collected into *pf2*, which is passed recursively to *splitByInst*.

> **Details:** *partition* takes a predicate and a list and returns a pair of lists: those elements that satisfy the predicate, and those that do not, respectively. *partition* is defined in the *List* Library as:
>
> ```
> partition :: (a → Bool) → [a] → ([a], [a])
> partition p xs =
>   foldr select ([], []) xs
>       where select x (ts, fs) | p x = (x : ts, fs)
>                               | otherwise = (ts, x : fs)
> ```

The crux of the conversion process is in *performToMEvs*, which converts a *Performance* into a stream of time-stamped messages, i.e. a stream of (*Tick*, *Message*) pairs:

```
type MEvent = (Ticks, Message)

defST = 500000

performToMEvs :: UserPatchMap → (InstrumentName, Performance) → [MEvent]
performToMEvs upm (inm, pf) =
  let (midiChan, progNum) = upmLookup upm inm
      setupInst = (0, ProgramChange midiChan progNum)
      setTempo = (0, TempoChange defST)
      loop [] = []
      loop (e : es) = let (mev1, mev2) = mkMEvents midiChan e
```

```
                        in mev1 : insertMEvent mev2 (loop es)
      in setupInst : setTempo : loop pf
```

A source of incompatibilty between Haskore and Midi is that Haskore represents notes with an onset and a duration, while Midi represents them as two separate events, a note-on event and a note-off event. Thus *MkMEvents* turns a Haskore *Event* into two *MEvents*, a *NoteOn* and a *NoteOff*.

```
mkMEvents :: Channel → Event → (MEvent, MEvent)
mkMEvents mChan (Event{eTime = t, ePitch = p, eDur = d, eVol = v})
                   = ((toDelta t, NoteOn mChan p v′),
                      (toDelta (t + d), NoteOff mChan p v′))
             where v′ = max 0 (min 127 (fromIntegral v))

toDelta t = round (t ∗ 4.0 ∗ fromIntegral division)
```

The time-stamp associated with an event in Midi is called a *delta-time*, and is the time at which the event should occur expressed in time-code divisions since the beginning of the performance. Since there are 96 time-code divisions per quarter note, there are 4 times that many in a whole note; multiplying that by the time-stamp on one of our *Event*s gives us the proper delta-time.

In the code for *performToMEvs*, note that the location of the first event returned from *mkMEvents* is obvious; it belongs just where it was created. However, the second event must be inserted into the proper place in the rest of the stream of events; there is no way to know of its proper position ahead of time. The function *insertMEvent* is thus used to insert an *MEvent* into an already time-ordered sequence of *MEvent*s.

```
insertMEvent :: MEvent → [MEvent] → [MEvent]
insertMEvent mev1 [] = [mev1]
insertMEvent mev1@(t1, _) mevs@(mev2@(t2, _) : mevs′) =
      if t1 ⩽ t2 then mev1 : mevs
                 else mev2 : insertMEvent mev1 mevs′
```

13.3 Putting It All Together

[TODO: Move the code from haskore.lhs to this section – i.e. the *PerformanceDefault* type class, the family of *play* functions, and so on.]

Chapter 14

Basic Input/Output

So far the only input/output (IO) that we have seen in Haskore is the use of the *play* function to generate the Midi output corresponding to a *Music* value. But we've said very little about the *play* function itself. What is its type? How does it work? How does one do IO in a purely functional language such as Haskell? Our goal in this chapter is to answer these questions. Then in the next chapter we will describe an elegant way to do IO involving a "graphical musical interface."

14.1 IO in Haskell

The Haskell Report defines the result of a program to be the value of the variable *main* in the module *Main*. On the other hand, as you know, the GHCi implementation of Haskell allows you to type whatever expression you wish to the command prompt, and it will evaluate it for you. In both cases, the Haskell system "executes a program" by evaluating an expression, which (for a well-behaved program) eventually yields a value. The system must then display that value on your computer screen in some way that makes sense to you. Most systems will try to display the result in the same way that you would type it as part of a program. So an integer is printed as an integer, a string as a string, a list as a list, and so on. We will refer to the area of the computer screen where this result is printed as the *standard output area*, which may vary from one implementation to another.

But what if a program is intended to write to a file? Or print a file on a printer? Or, the main topic of this book, to play some music through the computer's sound card, or an external Midi device? These are examples of *output*, and there are related questions about *input*: for example, how does a program receive input from the computer keyboard or mouse, or receive input from a Midi keyboard?

In general, how does Haskell's "expression-oriented" notion of "computation by calculation" accommodate these various kinds of input and output?

The answer is fairly simple: in Haskell there is a special kind of value called an *action*. When a Haskell system evaluates an expression that yields an action, it knows not to try to display the result in the standard output area, but rather to "take the appropriate action." There are primitive actions—such as writing a single character to a file or receiving a single character from a Midi keyboard—as well as compound actions—such as printing an entire string to a file or playing an entire piece of music. Haskell expressions that evaluate to actions are commonly called *commands*.

Commands are still just expressions, of course, and some commands return a value for subsequent use by the program: keyboard input, for instance. A command that returns a value of type T has type $IO\ T$; if no useful value is returned the command has type $IO\ ()$. The simplest example of a command is $return\ x$, which for a value $x :: T$ immediately returns x and has type $IO\ T$.

> **Details:** The type () is called the *unit type*, and has exactly one value, which is also written (). Thus $return\ ()$ has type $IO\ ()$, and is often called a "noop" because it is an operation that does nothing and returns no useful result. Despite the negative connotation, it is used quite often!

Remember that all expressions in Haskell must be well-typed before a program is run, so a Haskell implementation knows ahead of time, by looking at the type, that it is evaluating a command, and is thus ready to "take action."

To make these ideas clearer, let's consider a few examples. One useful IO command is *putStr*, which prints a string argument to the standard output area, and has type $String \rightarrow IO\ ()$. The () simply indicates that there is no useful result returned from this action; its sole purpose is to print its argument to the standard output area. So the program:

```
module Main where
main = putStr "Hello World\n"
```

is the canonical "Hello World" program that is often the first program that people write in a new language.

Suppose now that we want to perform *two* actions, such as first writing to a file named `"testFile.txt"`, then printing to the standard output area. Haskell has a special keyword, **do**, to denote the beginning of a sequence of commands such as this, and so we can write:

```
do writeFile "testFile.txt" "Hello File System"
   putStr "Hello World\n"
```

where the file-writing function *writeFile* has type:

```
writeFile :: FilePath → String → IO ()
type FilePath = String
```

> **Details:** A **do** expression allows us to sequence an arbitrary number of commands, each of type *IO* (), using layout to distinguish them (just as in a **let** or **where** expression). When used in this way, the result of a **do** expression also has type *IO* ().

So far we have only used actions having type *IO* (); i.e. output actions. But what about input? As above, we will consider input from both the user and the file system.

To receive a line of input from the user (which will be typed in the *standard input area* of the computer screen, usually the same as the standard output area) we can use the function:

```
getLine :: IO String
```

Suppose, for example, that we wish to read a line of input using this function, and then write that line (a string) to a file. To do this we write the compound command:

```
do s ← getLine
   writeFile "testFile.txt" s
```

Note the syntax for binding *s* to the result of executing the *getLine* command; since the type of *getLine* is *IO String*, the type of *s* is *String*. Its value is then used in the next line as an argument to the *writeFile* command.

Similarly, we can read the entire contents of a file using the command *readFile* :: *FilePath* → *IO String*. For example:

```
do s ← readFile "testFile.txt"
   putStr s
```

There are many other commands available for file, system, and user IO, some in the Standard Prelude, and some in various libararies (such as *IO*, *Directory*, *System*, and *Time*). We will not discuss any of these here; rather, in the next chapter will concentrate on Midi input as well as a collection of graphical input widgets (such as sliders and pushbuttons) that we collectively refer to as Haskore's *graphical musical interface*.

Before that, however, we wish to emphasize that, despite the special **do** syntax, Haskell's IO commands are no different in status from any other Haskell function or value. For example, it is possible to create a *list* of actions, such as:

```
actionList = [putStr "Hello World\n",
              writeFile "testFile.txt" "Hello File System",
              putStr "File successfully written."]
```

However, a list of actions is just a list of values: they actually don't *do* anything until they are sequenced appropriately using a **do** expression, and then returned as the value *main* of the overall program. Still, it is often convenient to place actions into a list as above, and the Haskell Report and Libraries have some useful functions for turning them into commands. In particular, the function *sequence_* in the Standard Prelude, when used with IO, has type:

```
sequence_ :: [IO a] → IO ()
```

and can thus be applied to the *actionList* above to yield the single command:

```
main = sequence_ actionList
```

For a more interesting example of this idea, recall that Haskell's strings are really just *lists of characters*. Indeed, *String* is a type synonym for a list of characters:

```
type String = [Char]
```

Because strings are used so often, Haskell allows you to write "Hello" instead of ['H', 'e', 'l', 'l', 'o']. But keep in mind that this is just syntax—strings really are just lists of characters, and these two ways of writing them are identical from Haskell's perspective.

(Earlier the type synonym *FilePath* was defined for *String*. This shows that type synonyms can be made for other type synonyms.)

Now back to the example. From the function *putChar* :: *Char* → *IO* (), which prints a single character to the standard output area, we can define the function *putStr* used earlier, which prints an entire string. To do this, let's first define a function that converts a list of characters (i.e. a string) into a list of IO actions:

```
putCharList :: String → [IO ()]
putCharList [] = []
putCharList (c : cs) = putChar c : putCharList cs
```

With this, *putStr* is easily defined:

```
putStr :: String → IO ()
putStr s = sequence_ (putCharList s)
```

Note that the expression *putCharList s* is a list of actions, and *sequence_* is used to turn them into a single (compound) command, just as we did earlier.

(*putStr* can also be defined directly as a recursive function, but we leave that as an exercise.)

IO processing in Haskell is consistent with everything we have learned about programming with expressions and reasoning through calculation, although that may not be completely obvious yet. Indeed, it turns out that a **do** expression is just syntax for a more primitive way of combining actions using functions, namely a *monad*. be revealed in full in Chapter **??**.

14.2 Reading and Writing Midi Files

[TODO: Explain Midi-file IO functions defined in *Codec.Midi*, as well as the Haskore functions for writing Midi files.]

Chapter 15

Graphical Music Interface

```
module GMI where
import Haskore
import Haskore.UI
```

Most music software packages have graphical user interfaces that provide varying degrees of functionality to the user. In Haskore a basic set of widgets is provided that are collectively referred to as the *graphical music interface*, or GMI. This interface has two levels of abstraction: At the *user interface (UI) level*, basic IO-like commands are provided for creating graphical sliders, pushbuttons, and so on for input, and textual displays and graphic images for output (in the future, other kinds of graphic input and output, including virtual keyboards, plots, and so on, will be provided). In addition to these graphical widgets, the UI level also provides an interface to standard Midi input and output devices.

The second level of abstraction of the GMI is the *signal level*. A *signal* is a time-varying quantity that nicely captures the behavior of many GUI widgets. A special case of a signal is an *event*, and a special case of an event is a *Midi event*, such as a Note-On or Note-Off message.

We begin our discussion with a description of signals and events.

15.1 Signals

A value of type *Signal T* is a time-varying value of type *T*. For example, *Signal Float* is a time-varying floating-point number, *Signal AbsPitch* is a time-varing absolute pitch, and so on. Abstractly, one can think of a signal as a function:

```
Signal a = Time → a
```

where *Time* is some suitable representation of time (currently *Double* in Haskore).

However, this is not how signals are actually implemented in Haskore, indeed the above is not even valid Haskell syntax. Nevertheless it is helpful to think of signals in this way. For pedagogical purposes, we can go one step further and write the above as a data declaration:

```
data Signal a = Sig (Time → a)
```

and then describe in more detail how signals are manipulated once this concrete representation is in hand.

For starters, one of the more common set of operations that we desire for signals is *arithmetic*. For example, we'd like to add, subtract, and multiply signals, as well as apply transcendental functions such as sine, cosine, and exponentiation. Haskell's numeric classes provide a delightfully convenient way to do this. For example, we can declare *Signal* to be an instance of the class *Num*:

```
instance Num a ⇒ Num (Signal a) where
  Sig f1 + Sig f2 = Sig (λt → f1 t + f2 t)
  Sig f1 ∗ Sig f2 = Sig (λt → f1 t ∗ f2 t)
  ...
```

Keep in mind that signals aren't actually implemented in this way, but conceptually this should give you an idea of the desired behavior.

More abstractly, the following functions can be used to "lift" static functions to the time-varying domain of signals:

```
lift0 :: a → Signal a
lift1 :: (a → b) → (Signal a → Signal b)
lift2 :: (a → b → c) → (Signal a → Signal b → Signal c)
lift3 :: (a → b → c → d) → (Signal a → Signal b → Signal c → Signal d)
```

So, for example, we could have written the above instance declaration like this:

```
instance Num a ⇒ Num (Signal a) where
  (+) = lift2 (+)
  (∗) = lift2 (∗)
  ...
```

Haskore also defines instances of *Signal* for the classes *Fractional* and *Floating*. For example:

```
instance Floating a ⇒ Floating (Signal a) where
  pi = lift0 pi
  sin = lift1 sin
  exp = lift1 exp
  ...
```

(See the Haskell Report for the full list of operations in these classes.)

Of course, any function can be lifted. For example, consider the *pitch* function from Chapter 2. Its type is *AbsPitch* → *Pitch*, therefore the function *lift1 pitch* must have type *Signal AbsPitch* → *Signal Pitch*. We will see a larger example using this idea shortly.

Sometimes we need to zip and unzip values at the signal level. The following functions facilitate this:

```
join :: Signal a → Signal b → Signal (a, b)
split :: Signal (a, b) → (Signal a, Signal b)
fstS :: Signal (a, b) → Signal a
sndS :: Signal (a, b) → Signal b
```

The behavior of these functions should be clear from their type signatures. For example, *join* takes two signals and zips their values together pointwise; *split* takes a signal of pairs and returns a pair of signals; and so on.

We would also like to compare signals, but it is not as easy to overload the relational operators as we did the arithmetic operators, since they do not have a uniform type structure, and are not members of a convenient type class. Therefore the following special operators are defined:

```
(<∗), (>∗), (⩽∗), (⩾∗) :: Ord a ⇒ Signal a → Signal a → Signal Bool
(==∗), (≠∗) :: Eq a ⇒ Signal a → Signal a → Signal Bool
(&&∗), (||∗) :: Signal Bool → Signal Bool → Signal Bool
notS :: Signal Bool → Signal Bool
```

For example, if *s1*, *s2* :: *Signal AbsPitch* are two signals of absolute pitches, then *s1* ==∗ *s2* is a signal of Boolean values that represents the pointwise equality comparison of the two signals.

15.1.1 Stateful Signals

Some signals are *stateful*, meaning that they depend on past values in some way. A particularly important example of a stateful signal is the *integral* of a signal, which can be computed with the following function:

$$integral :: Signal\ Time \rightarrow Signal\ Double \rightarrow Signal\ Double$$

The first argument to *integral* is a signal that represents the current time. We will say more about this in the next section, but conceptually you can think of *integral t s* as the integral of *s* with respect to *t*.

Another kind of stateful signal can be generated with the functions:

$$initS, initS' :: a \rightarrow Signal\ a \rightarrow Signal\ a$$

initS conceptually introduces an infinitesimally small delay in a signal by "initializing" it with a given value. In other words, the signal *initS v s* behaves just like *s*, but it has the value *v* at time 0, and takes on the values of *s* henceforth. *initS′ v s* behaves similarly, except that it *replaces* the initial value in *s* with *v*. In the limit, these are mathematically the same, but in practice, there is sometimes reason to choose one over the other.

As an example of the use of *initS*, suppose we have a signal $s :: Signal\ AbsPitch$, and we wish to know when its value changes—i.e. we would like a value of type *Signal Bool* that is *True* just at those moments when *s* has changed.[1] Using *initS* we can compute the desired result by comparing the value of the signal to its values an infinitesimally short time in the past; that is:

$$s \neq\!* \ initS\ 0\ s$$

15.2 Events and Reactivity

Although signals are a nice abstraction of time-varying entities, and the world is arguably full of such entities, there are some things that happen at discrete points in time, like a mouse click, or a Midi keyboard press, and so on. We call these *events*. To represent events, and have them coexist with signals, recall the *Maybe* type defined in the Standard Prelude:

$$\textbf{data}\ Maybe\ a = Nothing \mid Just\ a$$

We define an event simply as a value of type *Signal* (*Maybe a*), and in this sense events in Haskore are really event *streams*, since more than one may occur over time. We say that the value associated with an event is "attached to" that event. For clarity we define *EventS* as a type synonym:

[1] Mathematically, if *s* were truly a continuous numeric function of time, this would be the same as asking when the derivative is non-zero. But we would also like to compute this result for integral types such as *AsbPitch*, as well as for non-numeric types.

$$\textbf{type}\ \mathit{EventS}\ a = \mathit{Signal}\ (\mathit{Maybe}\ a)$$

"*EventS*" can be read either as "event stream" or "event signal," although we will often just write "event."

There are lots of things that we would like to do with events. For example, to map a function over the values attached to an event stream, we can use:

$$(=\!\!\gg) :: \mathit{EventS}\ a \rightarrow (a \rightarrow b) \rightarrow \mathit{EventS}\ b$$

For convenience we also define a version of $(=\!\!\gg)$ that ignores its input value:

$$\begin{aligned}&(-\!\!\gg) :: \mathit{EventS}\ a \rightarrow b \rightarrow \mathit{EventS}\ b\\ &\mathit{s1} -\!\!\gg v = s =\!\!\gg (\lambda\_ \rightarrow v)\end{aligned}$$

Mnemonically, as expression of the form $e =\!\!\gg f$ can be read as "send the event stream e through the function f."

We can merge two event streams using:

$$(.|.) :: \mathit{EventS}\ a \rightarrow \mathit{EventS}\ a \rightarrow \mathit{EventS}\ a$$

If two events happen at the same time, preference is given to the one in the first argument.

Another useful operation is turning a Boolean signal into an event stream, which can be done in two different ways:

$$\mathit{edge}, \mathit{when} :: \mathit{Signal}\ \mathit{Bool} \rightarrow \mathit{EventS}\ ()$$

edge s generates an event whenever the Boolean signal *s* changes from *False* to *True*—in signal processing this is called an "edge detector," and thus the name. *when s* is also an edge detector, but it generates an event whenever *s* changes either from *False* to *True* or from *True* to *False*.

A related operation is:

$$\mathit{unique} :: \mathit{Eq}\ a \Rightarrow \mathit{Signal}\ a \rightarrow \mathit{EventS}\ a$$

which generates an event whenever the signal argument changes, and attaches the value of the signal at that time to the event. For example, if $\mathit{ap} :: \mathit{Signal}\ \mathit{AbsPitch}$ changes its pitch once every second, starting with absolute pitch 0 and incrementally moving upward, then *unique ap* will generate an event stream whose attached values are successively 0, 1, 2, and so on. Furthermore, the signal:

$$\mathit{unique}\ \mathit{ap} =\!\!\gg \mathit{lift1}\ \mathit{pitch}$$

$$\begin{aligned}
&\mathit{snapshot} :: \mathit{EventS}\ a \rightarrow \mathit{Signal}\ b \rightarrow \mathit{EventS}\ (a, b)\\
&\mathit{snapshot\_} :: \mathit{EventS}\ a \rightarrow \mathit{Signal}\ b \rightarrow \mathit{EventS}\ b\\
&\mathit{hold} :: a \rightarrow \mathit{EventS}\ a \rightarrow \mathit{Signal}\ a\\
&\mathit{accum} :: a \rightarrow \mathit{EventS}\ (a \rightarrow a) \rightarrow \mathit{Signal}\ a
\end{aligned}$$

Table 15.1: Signal Samplers

generates events once per second, with the attached values being the pitches $(C, 0)$, $(Cs, 0)$, $(D, 0)$, and so on.

The useful collection of functions shown in Table 15.1 can be thought of as "signal samplers." For example, if $\mathit{ticks} :: \mathit{EventS}\ ()$ generates unit events at some sampling rate, then $\mathit{snapshot\_}\ \mathit{ticks}\ \mathit{ap}$ is a stream of events at the same rate, but the value attached to each event is the value of the absolute pitch *ap* at that time. *snapshot* behaves similarly, but pairs the sampled value with the original event value.

$\mathit{hold}\ v\ e$ is a signal whose initial value is v, which it "holds" until the first event in e happens, at which point it changes to the value attached to that event, which it then "holds" until the next event, and so on. *accum* is a bit like *scan*. The signal $\mathit{accum}\ v\ e$ starts with the value v, but then applies the function attached to the first event to that value to get the next value, and so on.

More generally, perhaps the most fundamental set of operations on events are the ones that introduce *reactivity*: the ability to change a signal's behavior in response to an event. There are two operations for this purpose:

$$\mathit{switch}, \mathit{untilS} :: \mathit{Signal}\ a \rightarrow \mathit{EventS}\ (\mathit{Signal}\ a) \rightarrow \mathit{Signal}\ a$$

The signal s ‘*untilS*‘ e initially behaves just like s, until the first event in e occurs. It then behaves forever after like the behavior attached to that event. s‘*switch*‘ e behaves similarly, except that each subsequent event after the first will change the behavior to the event's new attached signal value.

15.3 The UI Level

It is at the UI level that "graphical widgets" are actually created, using a style very similar to the way we did IO in Chapter 14. But instead of values of type *IO T*, which we referred to as *IO actions*, we will use values of type *UI T*, which we refer to as *UI actions*. Just like *IO*, the *UI* type is fully abstract, and is an instance of *Monad*, which allows us to use the **do** syntax.

$$\begin{array}{l}
\mathit{label} :: \mathit{String} \rightarrow \mathit{UI}\ ()\\
\mathit{display} :: \mathit{Signal}\ \mathit{String} \rightarrow \mathit{UI}\ ()\\
\mathit{button} :: \mathit{String} \rightarrow \mathit{UI}\ (\mathit{Signal}\ \mathit{Bool})\\
\mathit{checkbox} :: \mathit{String} \rightarrow \mathit{Bool} \rightarrow \mathit{UI}\ (\mathit{Signal}\ \mathit{Bool})\\
\mathit{radio} :: [\mathit{String}] \rightarrow \mathit{Int} \rightarrow \mathit{UI}\ (\mathit{Signal}\ \mathit{Int})\\
\mathit{hSlider}, \mathit{vSlider} :: (\mathit{RealFrac}\ a) \Rightarrow (a, a) \rightarrow a \rightarrow \mathit{UI}\ (\mathit{Signal}\ a)\\
\mathit{hiSlider}, \mathit{viSlider} :: (\mathit{Integral}\ a) \Rightarrow a \rightarrow (a, a) \rightarrow a \rightarrow \mathit{UI}\ (\mathit{Signal}\ a)\\
\mathit{canvas} :: \mathit{Dimension} \rightarrow \mathit{EventS}\ \mathit{Graphic} \rightarrow \mathit{UI}\ ()
\end{array}$$

Table 15.2: GMI Input Widgets

15.3.1 Input Widgets

Haskore's graphical input widgets are shown in Table 15.2. The names and type signatures of these functions suggest their functionality.

For example, a simple (static) text string can be displayed using:

$$\mathit{label} :: \mathit{String} \rightarrow \mathit{UI}\ ()$$

Alternatively, a time-varying string can be be displayed using:

$$\mathit{display} :: \mathit{Signal}\ \mathit{String} \rightarrow \mathit{UI}\ ()$$

button, *checkbox*, and *radio* are three kinds of pushbuttons. A *button* or *checkbox* is pressed and unpressed independently of others, whereas *radio* buttons are dependent upon each other—specifically, only one can be "on" at a time, so pressing one will turn off the others. The string argument to these functions is the label attached to the button. *radio* takes a list of strings, each being the label of one of the buttons in the mutually-exclusive group; indeed the length of the list determines how many buttons are in the group.

hSlider, *vSlider*, *hiSlider* and *viSlider* are four kinds of sliders—the first two yield floating-point numbers in a given range, oriented horizontally and vertically, respectively, whereas the latter two return integral numbers. For the integral sliders, the first argument is the size of the step taken when the slider is clicked at any point on either side of the slider "handle." In each of the four cases, the other two arguments are the range and initial setting of the slider, respectively.

As a larger example, let's combine our running example of absolute pitches with a slider. We will define a UI program that has a single slider representing the absolute pitch, and a display widget that displays the pitch corresponding to the setting of the slider:

```
title :: String → UI a → UI a
setSize :: Dimension → UI a → UI a
pad :: (Int, Int, Int, Int) → UI a → UI a
topDown, bottomUp, leftRight, rightLeft :: UI a → UI a
```

Table 15.3: GMI Layout Widget Transformers

```
ui1 = do ap ← title "Absolute Pitch" (hiSlider 1 (0, 100) 0)
         title "Pitch" (display (lift1 (show ∘ pitch) ap))
```

Note how the use of signals makes this dynamic UI trivial to write.

[TODO: explain canvas]

15.3.2 UI Transformers

Table 15.3 shows a set of "UI transformers"—functions that take UI values as input, and return modified UI's as output.

title attaches a title (a string) to a UI, and *setSize* establishes a fixed size for a UI. *pad* (w, n, e, s) *ui* adds w pixels of space to the "west" of the UI *ui*, and n, e, and s pixels of space to the north, east, and south, respectively. The other four functions are used to control the relative layout of the widgets within a UI. By default widgets are arranged top-to-bottom, but, for example, we could modify the previous UI program to arrange the two widgets left-to-right:

```
ui2 = leftRight$
        do ap ← title "Absolute Pitch" (hiSlider 1 (0, 100) 0)
           title "Pitch" (display (lift1 (show ∘ pitch) ap))
```

Layout tranformers can be nested, so a fair amount of flexibility is available.

15.3.3 Midi Input and Output

Here are the widgets for Midi input and output:

```
midiIn :: Signal DeviceID → UI (EventS [MidiMessage])
midiOut :: Signal DeviceID → EventS [MidiMessage] → UI ()
```

Except for the *DeviceID* (about which more will be said shortly), these functions are fairly straigtforward: *midiOut* takes a stream of *MidiMessage* events and sends them to the Midi output device, whereas *midiIn* generates a stream of *MidiMessage* events corresponding to the messages sent by the Midi input device. The *MidiMessage* data type is defined as:

```
data MidiMessage = ANote{ channel :: Channel, key :: Key,
                          velocity :: Velocity, duration :: Time }
                 | Std Message
     deriving Show
```

The *Message* data type was described in Chapter 13, and is defined in the *Codec.Midi* module. The additional *ANote* message above allows one to specify a note with duration. Such a message is conveniently transformed “behind-the-scenes” into a Note-On and Note-Off message sequence.

As an example of the use of *midiOut*, we will modify our previous UI program to output an *ANote* message everytime the absolute pitch changes:

```
ui3 = do ap ← title "Absolute Pitch" (hiSlider 1 (0, 100) 0)
         title "Pitch" (display (lift1 (show ∘ pitch) ap))
         let ns = unique ap =≫ (λk → [ANote 0 k 100 0.1])
         midiOut 0 ns
```

Note the use of *unique* to generate an event when the pitch changes, and the use of (=≫) to convert those events into *ANote*s.

Also note that the *DeviceID* argument to *midiOut* is set to 0. The Midi device ID is a system-dependent concept that provides an operating system with a simple way to uniquely identify various Midi devices that may be attached to a computer. Indeed, as devices are dynamically connected and disconnected from a computer, the mapping of these IDs to a particular device may change. If you try to run the above code, it may or may not work, depending on whether the Midi device with ID 0 corresponds to the preferred Midi output device on your machine.

To overcome this problem, most Midi software programs allow the user to select the preferred Midi input and output devices. The user usually has the best knowledge of which devices are connected, and which devices to use. In Haskore, the easiest way to do this is using the UI widgets:

```
selectInput, selectOutput :: UI (Signal DeviceID)
```

Each of these widgets automatically queries the operating system to obtain a list of connected Midi devices, and then displays the list as a set of radio buttons, thus allowing the user to select one of them. Note that the result is a signal, and the *DeviceID* arguments to *midiIn* and *midiOut* are also signals. This makes wiring up the user choice very easy. For example, we can modify the previous program to look like this:

```
ui4 = do devid ← selectOutput
         ap ← title "Absolute Pitch" (hiSlider 1 (0, 100) 0)
         title "Pitch" (display (lift1 (show ∘ pitch) ap))
         let ns = unique ap =≫ (λk → [ANote 0 k 100 0.1])
         midiOut devid ns
```

For an example using input as well, here is a simple program that copies each Midi message verbatim from the selected input device to the selected output device:

```
ui5 = do mi ← selectInput
         mo ← selectOutput
         m ← midiIn mi
         midiOut mo m
```

15.3.4 Timer Widgets

Remember that there is no "hidden" time in the GMI—anything that depends on the notion of time (such as *integral* discussed earlier) takes a time signal explicitly as an argument. For this purpose, the following function generates a signal corresponding to the current time:

```
time :: UI (Signal Time)
```

Besides *integral*, another function that depends explicitly on time is the following, which creates a *timer*:

```
timer :: Signal Time → Signal Double → EventS ()
```

timer t i takes a time source *t* and a signal *i* that represents the timer interval (in seconds), and generates a stream of events, with each pair of consecutive events separated by the timer interval. Note that the timer interval is itself a signal, so the timer output can have varying frequency.

As an example of this, let's modify our previous UI so that, instead of playing a note everytime the absolute pitch changes, we will output a note continuously, at a rate controlled by a second slider:

```
ui6 = do devid ← selectOutput
         ap ← title "Absolute Pitch" (hiSlider 1 (0, 100) 0)
         title "Pitch" (display (lift1 (show ∘ pitch) ap))
         t ← time
         f ← title "Tempo" (hSlider (1, 10) 1)
         let ticks = timer t (1 / f)
         let ns = snapshot_ ticks ap =≫ (λk → [ANote 0 k 100 0.1])
         midiOut devid ns
```

Note that:

- The time *t* is needed solely to drive the timer.
- The rate of *ticks* is controlled by the slider. A higher slider value causes a lower time between ticks, and thus a higher frequency, or tempo.
- *snapshot_* uses the timer output to control the sample rate of the absolute pitch.

Finally, an event stream can be delayed by a given (variable) amount of time using the following function:

```
delayt :: Signal Time → Signal Double → EventS a → EventS a
```

The second argument specifies the amount of delay to be applied to the third argument.

15.4 Putting It All Together

Recall that a Haskell program must eventually be a value of type *IO* (), and thus we need a function to turn a *UI* value into a *IO* value—i.e. the UI needs to be "run." We can do this using one of the following two functions:

```
runUI :: String → UI a → IO ()
runUIEx :: Dimension → String → UI a → IO ()
```

Both of these functions take a string argument that is displayed in the title bar of the graphical window that is generated. *runUIEx* additionally takes the dimensions of the window as an argument.

Executing *runUI s ui* or *runUIEx d s ui* will create a single GUI window whose behavior is governed by the argument *ui* :: *UI a*. From the previous examples we can now generate the following executable programs:

```
main1 = runUI "Simple UI (default layout)" ui1
main2 = runUI "Simple UI (left-to-right layout)" ui2
main3 = runUI "Pitch Player" ui3
main4 = runUI "Pitch Player with Midi Device Selection" ui4
main5 = runUI "Midi Input / Output UI" ui5
main6 = runUI "Pitch Player with Timer" ui6
```

15.5 Examples

TODO: Provide some larger examples. For now, see the files:

```
haskore/src/example/UIExamples.hs
haskore/src/example/GMIExamples.hs
```

Appendix A

The PreludeList Module

The use of lists is particularly common when programming in Haskell, and thus, not surprisingly, there are many pre-defined polymorphic functions for lists. The list data type itself, plus some of the most useful functions on it, are contained in the Standard Prelude's *PreludeList* module, which we will look at in detail in this chapter. There is also a Standard Library module called *List* that has additional useful functions. It is a good idea to become familiar with both modules.

Although this chapter may feel like a long list of "Haskell features," the functions described here capture many common patterns of list usage that have been discovered by functional programmers over many years of trials and tribulations. In many ways higher-order declarative programming with lists takes the place of lower-level imperative control structures in more conventional languages. By becoming familiar with these list functions you will be able to more quickly and confidently develop your own applications using lists. Furthermore, if all of us do this, we will have a common vocabulary with which to understand each others' programs. Finally, by reading through the code in this module you will develop a good feel for how to write proper function definitions in Haskell.

It is not necessary for you to understand the details of every function, but you should try to get a sense for what is available so that you can return later when your programming needs demand it. In the long run you are well-advised to read the rest of the Standard Prelude as well as the various Standard Libraries, to discover a host of other functions and data types that you might someday find useful in your own work.

A.1 The PreludeList Module

To get a feel for the *PreludeList* module, let's first look at its module declaration:

```
module PreludeList (
    map, (⧺), filter, concat,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3)
  where

import qualified Char (isSpace)

infixl 9!!
infixr 5⧺
infix 4 ∈, ∉
```

We will not discuss all of the functions listed above, but will cover most of them (and some were discussed in previous chapters).

A.2 Simple List Selector Functions

head and *tail* extract the first element and remaining elements, respectively, from a list, which must be non-empty. *last* and *init* are the dual functions that work from the end of a list, rather than from the beginning.

```
head :: [a] → a
head (x : _) = x
head [] = error "PreludeList.head: empty list"

last :: [a] → a
last [x] = x
last (_ : xs) = last xs
last [] = error "PreludeList.last: empty list"
```

```
tail :: [a] → [a]
tail (_ : xs) = xs
tail [] = error "PreludeList.tail: empty list"

init :: [a] → [a]
init [x] = []
init (x : xs) = x : init xs
init [] = error "PreludeList.init: empty list"
```

Although *head* and *tail* were previously discussed in Section 3.1, the definitions here include an equation describing their behaviors under erroneous situations—such as selecting the head of an empty list—in which case the *error* function is called. It is a good idea to include such an equation for any definition in which you have not covered every possible case in pattern-matching; i.e. if it is possible that the pattern-matching could "run off the end" of the set of equations. The string argument that you supply to the *error* function should be detailed enough that you can easily track down the precise location of the error in your program.

> **Details:** If such an error equation is omitted, and then during pattern-matching all equations fail, most Haskell systems will invoke the *error* function anyway, but most likely with a string that will be less informative than one you can supply on your own.

The *null* function tests to see if a list is empty.

```
null :: [a] → Bool
null [] = True
null (_ : _) = False
```

A.3 Index-Based Selector Functions

To select the nth element from a list, with the first element being the 0th element, we can use the indexing function (!!):

```
(!!) :: [a] → Int → a
(x : _) !! 0 = x
(_ : xs) !! n | n > 0 = xs !! (n − 1)
(_ : _) !! _ = error "PreludeList.!!: negative index"
[] !! _ = error "PreludeList.!!: index too large"
```

> **Details:** Note the definition of two error conditions; be sure that you understand under what conditions these two equations would succeed. In particular, recall that equations are matched in top-down order: the first to match is the one that is chosen.

take n xs returns the prefix of *xs* of length *n*, or *xs* itself if $n > length\ xs$. Similarly, *drop n xs* returns the suffix of *xs* after the first *n* elements, or [ ] if $n > length\ xs$. Finally, *splitAt n xs* is equivalent to (*take n xs*, *drop n xs*).

```
take :: Int → [a] → [a]
take 0 _ = [ ]
take _ [ ] = [ ]
take n (x : xs) | n > 0 = x : take (n − 1) xs
take _ _ = error "PreludeList.take: negative argument"

drop :: Int → [a] → [a]
drop 0 xs = xs
drop _ [ ] = [ ]
drop n (_ : xs) | n > 0 = drop (n − 1) xs
drop _ _ = error "PreludeList.drop: negative argument"

splitAt :: Int → [a] → ([a], [a])
splitAt 0 xs = ([ ], xs)
splitAt _ [ ] = ([ ], [ ])
splitAt n (x : xs) | n > 0 = (x : xs′, xs″)
                   where (xs′, xs″) = splitAt (n − 1) xs
splitAt _ _ = error "PreludeList.splitAt: negative argument"

length :: [a] → Int
length [ ] = 0
length (_ : l) = 1 + length l
```

For example:

```
take 3 [0, 1 . . 5] ⇒ [0, 1, 2]
drop 3 [0, 1 . . 5] ⇒ [3, 4, 5]
splitAt 3 [0, 1 . . 5] ⇒ ([0, 1, 2], [3, 4, 5])
```

A.4 Predicate-Based Selector Functions

takeWhile p xs returns the longest (possibly empty) prefix of *xs*, all of whose elements satisfy the predicate *p*. *dropWhile p xs* returns the remaining suffix. Finally, *span p xs* is equivalent to (*takeWhile p xs*, *dropWhile p xs*), while *break p* uses the negation of *p*.

$$\begin{array}{l}
\mathit{takeWhile} :: (a \to \mathit{Bool}) \to [\,a\,] \to [\,a\,]\\
\mathit{takeWhile}\ p\ [\,] = [\,]\\
\mathit{takeWhile}\ p\ (x : \mathit{xs})\\
\qquad\qquad\qquad \mid p\ x = x : \mathit{takeWhile}\ p\ \mathit{xs}\\
\qquad\qquad\qquad \mid \mathit{otherwise} = [\,]
\end{array}$$

$$\begin{array}{l}
\mathit{dropWhile} :: (a \to \mathit{Bool}) \to [\,a\,] \to [\,a\,]\\
\mathit{dropWhile}\ p\ [\,] = [\,]\\
\mathit{dropWhile}\ p\ \mathit{xs}@(x : \mathit{xs}')\\
\qquad\qquad\qquad \mid p\ x = \mathit{dropWhile}\ p\ \mathit{xs}'\\
\qquad\qquad\qquad \mid \mathit{otherwise} = \mathit{xs}
\end{array}$$

$$\begin{array}{l}
\mathit{span}, \mathit{break} :: (a \to \mathit{Bool}) \to [\,a\,] \to ([\,a\,], [\,a\,])\\
\mathit{span}\ p\ [\,] = ([\,], [\,])\\
\mathit{span}\ p\ \mathit{xs}@(x : \mathit{xs}')\\
\qquad\qquad\qquad \mid p\ x = (x : \mathit{xs}', \mathit{xs}'')\ \mathbf{where}\ (\mathit{xs}', \mathit{xs}'') = \mathit{span}\ p\ \mathit{xs}\\
\qquad\qquad\qquad \mid \mathit{otherwise} = (\mathit{xs}, [\,])
\end{array}$$

$$\mathit{break}\ p = \mathit{span}\ (\neg \circ p)$$

filter removes all elements not satisfying a predicate:

$$\begin{array}{l}
\mathit{filter} :: (a \to \mathit{Bool}) \to [\,a\,] \to [\,a\,]\\
\mathit{filter}\ p\ [\,] = [\,]\\
\mathit{filter}\ p\ (x : \mathit{xs}) \mid p\ x = x : \mathit{filter}\ p\ \mathit{xs}\\
\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{filter}\ p\ \mathit{xs}
\end{array}$$

A.5 Fold-like Functions

foldl1 and *foldr1* are variants of *foldl* and *foldr* that have no starting value argument, and thus must be applied to non-empty lists.

$$\begin{array}{l}
\mathit{foldl} :: (a \to b \to a) \to a \to [\,b\,] \to a\\
\mathit{foldl}\ f\ z\ [\,] = z\\
\mathit{foldl}\ f\ z\ (x : \mathit{xs}) = \mathit{foldl}\ f\ (f\ z\ x)\ \mathit{xs}
\end{array}$$

$$\begin{array}{l}
\mathit{foldl1} :: (a \to a \to a) \to [\,a\,] \to a\\
\mathit{foldl1}\ f\ (x : \mathit{xs}) = \mathit{foldl}\ f\ x\ \mathit{xs}
\end{array}$$

```
foldl1 _ [ ] = error "PreludeList.foldl1: empty list"

foldr :: (a → b → b) → b → [a] → b
foldr f z [ ] = z
foldr f z (x : xs) = f x (foldr f z xs)

foldr1 :: (a → a → a) → [a] → a
foldr1 f [x] = x
foldr1 f (x : xs) = f x (foldr1 f xs)
foldr1 _ [ ] = error "PreludeList.foldr1: empty list"
```

foldl1 and *foldr1* are best used in cases where an empty list makes no sense for the application. For example, computing the maximum or mimimum element of a list does not make sense if the list is empty. Thus *foldl1 max* is a proper function to compute the maximum element of a list.

scanl is similar to *foldl*, but returns a list of successive reduced values from the left:

```
scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]
```

For example:

```
scanl (+) 0 [1, 2, 3] ⇒ [0, 1, 3, 6]
```

Note that *last* (*scanl f z xs*) = *foldl f z xs*. *scanl1* is similar, but without the starting element:

```
scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]
```

Here are the full definitions:

```
scanl :: (a → b → a) → a → [b] → [a]
scanl f q xs = q : (case xs of
                         [ ] → [ ]
                         x : xs → scanl f (f q x) xs)
scanl1 :: (a → a → a) → [a] → [a]
scanl1 f (x : xs) = scanl f x xs
scanl1 _ [ ] = error "PreludeList.scanl1: empty list"

scanr :: (a → b → b) → b → [a] → [b]
scanr f q0 [ ] = [q0]
scanr f q0 (x : xs) = f x q : qs
                where qs@(q : _) = scanr f q0 xs
```

```
scanr1 :: (a → a → a) → [a] → [a]
scanr1 f [x] = [x]
scanr1 f (x : xs) = f x q : qs
                    where qs@(q : _) = scanr1 f xs
scanr1 _ [] = error "PreludeList.scanr1: empty list"
```

A.6 List Generators

There are some functions which are very useful for generating lists from scratch in interesting ways. To start, *iterate f x* returns an *infinite list* of repeated applications of f to x. That is:

```
iterate f x ⇒ [x, f x, f (f x), ...]
```

The "infinite" nature of this list may at first seem alarming, but in fact is one of the more powerful and useful features of Haskell.

[say more]

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

repeat x is an infinite list, with x the value of every element. *replicate n x* is a list of length n with x the value of every element. And *cycle* ties a finite list into a circular one, or equivalently, the infinite repetition of the original list.

```
repeat :: a → [a]
repeat x = xs where xs = x : xs

replicate :: Int → a → [a]
replicate n x = take n (repeat x)

cycle :: [a] → [a]
cycle [] = error "Prelude.cycle: empty list"
cycle xs = xs' where xs' = xs ++ xs'
```

A.7 String-Based Functions

Recall that strings in Haskell are just lists of characters. Manipulating strings (i.e. text) is a very common practice, so it makes sense that Haskell would have a few pre-defined functions to make this easier for you.

lines breaks a string at every newline character (written as `'\n'` in Haskell), thus yielding a *list* of strings, each of which contains no newline characters. Similary, *words* breaks a string up into a list of words, which were delimited by white space. Finally, *unlines* and *unwords* are the inverse operations: *unlines* joins lines with terminating newline characters, and *unwords* joins words with separating spaces. (Because of the potential presence of multiple spaces and newline characters, however, these pairs of functions are not true inverses of each other.)

```
lines :: String → [String]
lines "" = []
lines s = let (l, s') = break (== '\n') s
                          in l : case s' of
                                      [] → []
                                      (_ : s'') → lines s''

words :: String → [String]
words s = case dropWhile Char.isSpace s of
                           "" → []
                           s' → w : words s''
                                where (w, s'') = break Char.isSpace s'

unlines :: [String] → String
unlines = concatMap (++"\n")

unwords :: [String] → String
unwords [] = ""
unwords ws = foldr1 (λw s → w ++ ' ' : s) ws
```

reverse reverses the elements in a finite list.

```
reverse :: [a] – [a]
reverse = foldl (flip (:)) []
```

A.8 Boolean List Functions

and and *or* compute the logical “and” and “or,” respectively, of all of the elements in a list of Boolean values.

```
and, or :: [Bool] → Bool
and = foldr (∧) True
or = foldr (∨) False
```

Applied to a predicate and a list, *any* determines if any element of the list satisfies the predicate. An analogous behavior holds for *all*.

```
any, all :: (a → Bool) → [a] → Bool
any p = or ∘ map p
all p = and ∘ map p
```

A.9 List Membership Functions

elem is the list membership predicate, usually written in infix form, e.g., $x \in xs$ (which is why it was given a fixity declaration at the beginning of the module). *notElem* is the negation of this function.

```
elem, notElem :: (Eq a) ⇒ a → [a] → Bool
elem x = any (== x)
notElem x = all (≠ x)
```

It is common to store "key/value" pairs in a list, and to access the list by finding the value associated with a given key (for this reason the list is often called an *association list*). The function *lookup* looks up a key in an association list, returning *Nothing* if it is not found, or *Just y* if *y* is the value associated with the key.

```
lookup :: (Eq a) ⇒ a → [(a, b)] → Maybe b
lookup key [] = Nothing
lookup key ((x, y) : xys)
      | key == x = Just y
      | otherwise = lookup key xys
```

A.10 Arithmetic on Lists

sum and *product* compute the sum and product, respectively, of a finite list of numbers.

```
sum, product :: (Num a) ⇒ [a] → a
sum = foldl (+) 0
product = foldl (∗) 1
```

maximum and *minimum* return the maximum and minimum value, respectively from a non-empty, finite list whose element type is ordered.

```
maximum, minimum :: (Ord a) ⇒ [a] → a
maximum [] = error "Prelude.maximum: empty list"
maximum xs = foldl1 max xs

minimum [] = error "Prelude.minimum: empty list"
minimum xs = foldl1 min xs
```

Note that even though *foldl1* is used in the definition, a test is made for the empty list to give an error message that more accurately reflects the source of the problem.

A.11 List Combining Functions

map and (++) were defined in previous chapters, but are repeated here for completeness:

```
map :: (a → b) → [a] → [a]
map f [] = []
map f (x : xs) = f x : map f xs

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

concat appends together a list of lists:

```
concat :: [[a]] → [a]
concat xss = foldr (++) [] xss
```

concatMap does what it says: it concatenates the result of mapping a function down a list.

```
concatMap :: (a → [b]) → [a] → [b]
concatMap f = concat ∘ map f
```

zip takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements of the longer list are discarded. *zip3* takes three lists and returns a list of triples. (“Zips” for larger tuples are contained in the List Library.)

```
zip :: [a] → [b] → [(a, b)]
zip = zipWith (,)
```

$$
\begin{array}{l}
\mathit{zip3} :: [\,a\,] \rightarrow [\,b\,] \rightarrow [\,c\,] \rightarrow [\,(a,b,c)\,] \\
\mathit{zip3} = \mathit{zipWith3}\ (,,)
\end{array}
$$

Details: The functions *(,)* and *(,,)* are the pairing and tripling functions, respectively:

$$
\begin{array}{l}
(,) \Rightarrow \lambda x\ y \rightarrow (x,y) \\
(,,) \Rightarrow \lambda x\ y\ z \rightarrow (x,y,z)
\end{array}
$$

The *zipWith* family generalises the *zip* and *map* families (or, in a sense, combines them) by applying a function (given as the first argument) to each pair (or triple, etc.) of values. For example, *zipWith* (+) is applied to two lists to produce the list of corresponding sums.

$$
\begin{array}{l}
\mathit{zipWith} :: (a \rightarrow b \rightarrow c) \rightarrow [\,a\,] \rightarrow [\,b\,] \rightarrow [\,c\,] \\
\mathit{zipWith}\ z\ (a : \mathit{as})\ (b : \mathit{bs}) \\
\qquad\qquad\qquad = z\ a\ b : \mathit{zipWith}\ z\ \mathit{as}\ \mathit{bs} \\
\mathit{zipWith}\ \_\ \_\ \_ = [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{zipWith3} :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow [\,a\,] \rightarrow [\,b\,] \rightarrow [\,c\,] \rightarrow [\,d\,] \\
\mathit{zipWith3}\ z\ (a : \mathit{as})\ (b : \mathit{bs})\ (c : \mathit{cs}) \\
\qquad\qquad\qquad = z\ a\ b\ c : \mathit{zipWith3}\ z\ \mathit{as}\ \mathit{bs}\ \mathit{cs} \\
\mathit{zipWith3}\ \_\ \_\ \_\ \_ = [\,]
\end{array}
$$

The following two functions perform the inverse operations of *zip* and *zip3*, respectively.

$$
\begin{array}{l}
\mathit{unzip} :: [\,(a,b)\,] \rightarrow ([\,a\,],[\,b\,]) \\
\mathit{unzip} = \mathit{foldr}\ (\lambda(a,b){\sim}(\mathit{as},\mathit{bs}) \rightarrow (a : \mathit{as}, b : \mathit{bs}))\ ([\,],[\,])
\end{array}
$$

$$
\begin{array}{l}
\mathit{unzip3} :: [\,(a,b,c)\,] \rightarrow ([\,a\,],[\,b\,],[\,c\,]) \\
\mathit{unzip3} = \mathit{foldr}\ (\lambda(a,b,c){\sim}(\mathit{as},\mathit{bs},\mathit{cs}) \rightarrow (a : \mathit{as}, b : \mathit{bs}, c : \mathit{cs})) \\
\qquad\qquad\quad ([\,],[\,],[\,])
\end{array}
$$

Appendix B

Haskell's Standard Type Classes

This provides a "tour" through the predefined standard type classes in Haskell, as was done for lists in Chapter A. We have simplified these classes somewhat by omitting some of the less interesting methods; the Haskell Report and Standard Library Report contain more complete descriptions.

B.1 The Ordered Class

The equality class *Eq* was defined precisely in Chapter 11, along with a simplified version of the class *Ord*. Here is its full specification of class *Ord*; note the many default methods.

```
class (Eq a) ⇒ Ord a where
    compare :: a → a → Ordering
    (<), (⩽), (⩾), (>) :: a → a → Bool
    max, min :: a → a → a

    compare x y
        | x == y = EQ
        | x ⩽ y = LT
        | otherwise = GT

    x ⩽ y = compare x y ≠ GT
    x < y = compare x y == LT
    x ⩾ y = compare x y ≠ LT
```

```
        x > y = compare x y == GT

        max x y
              | x ⩾ y = x
              | otherwise = y
        min x y
              | x < y = x
              | otherwise = y
    data Ordering = LT | EQ | GT
          deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

Note that the default method for *compare* is defined in terms of (⩽), and that the default method for (⩽) is defined in terms of *compare*. This means that an instance of *Ord* should contain a method for at least one of these for everything to be well defined. (Using *compare* can be more efficient for complex types.) This is a common idea in designing a type class.

B.2 The Enumeration Class

Class *Enum* has a set of operations that underlie the syntactic sugar of *arithmetic sequences*; for example, the arithmetic sequence [1, 3 . .] is actually shorthand for *enumFromThen* 1 3. If this is true, then we should be able to generate arithmetic sequences for any type that is an instance of *Enum*. This includes not only most numeric types, but also *Char*, so that, for instance, `['a'..'z']` denotes the list of lower-case letters in alphabetical order. Furthermore, a user-defined enumerated type such as *Color*:

```
    data Color = Red | Orange | Yellow | Green | Blue | Indigo | Violet
```

can easily be given an *Enum* instance declaration, after which we can calculate the following results:

```
    [Red .. Violet] ⟹ [Red, Orange, Yellow, Green, Blue, Indigo, Violet]
    [Red, Yellow ..] ⟹ [Red, Yellow, Blue, Violet]
    fromEnum Green ⟹ 3
    toEnum 5 :: Color ⟹ Indigo
```

Indeed, the derived instance will give this result. Note that the sequences are still *arithmetic* in the sense that the increment between values is constant, even though the values are not numbers.

The complete definition of the *Enum* class is given below:

```
class Enum a where
    succ, pred :: a → a
    toEnum :: Int → a
    fromEnum :: a → Int
    enumFrom :: a → [a]      -- [n..]
    enumFromThen :: a → a → [a]      -- [n,n'..]
    enumFromTo :: a → a → [a]      -- [n..m]
    enumFromThenTo :: a → a → a → [a]      -- [n,n'..m]

        -- Minimal complete definition: toEnum, fromEnum
    succ = toEnum ∘ (+1) ∘ fromEnum
    pred = toEnum ∘ (subtract 1) ∘ fromEnum
    enumFrom x = map toEnum [fromEnum x ..]
    enumFromThen x y = map toEnum [fromEnum x, fromEnum y ..]
    enumFromTo x y = map toEnum [fromEnum x .. fromEnum y]
    enumFromThenTo x y z =
                    map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

The six default methods are sufficient for most applications, so when writing your own instance declaration it is usually sufficient to only provide methods for the remaining two operations: *toEnum* and *fromEnum*.

In terms of arithmetic sequences, the expressions on the left below are equivalent to those on the right:

enumFrom n	$[n\,..]$
enumFromThen n n′	$[n, n'\,..]$
enumFromTo n m	$[n\,..\,m]$
enumFromThenTo n n′ m	$[n, n'\,..\,m]$

B.3 The Bounded Class

The class *Bounded* captures data types that are linearly bounded in some way; i.e. they have both a minimum value and a maximum value.

```
class Bounded a where
    minBound :: a
    maxBound :: a
```

B.4 The Show Class

Instances of the class *Show* are those types that can be converted to character strings. This is useful, for example, when writing a representation of a value to the standard output area or to a file. The class *Read* works in the other direction: it provides operations for parsing character strings to obtain the values that they represent. In this section we will look at the *Show* class; in the next we will look at *Read*.

For efficiency reasons the primitive operations in these classes are somewhat esoteric, but they provide good lessons in both algorithm and software design, so we will look at them in some detail.

First, let's look at one of the higher-level functions that is defined in terms of the lower-level primitives:

$$\mathit{show} :: (\mathit{Show}\ a) \Rightarrow a \rightarrow \mathit{String}$$

Naturally enough, *show* takes a value of any type that is a member of *Show*, and returns its representation as a string. For example, *show* $(2 + 2)$ yields the string `"4"`, as does *show* $(6-2)$ and *show* applied to any other expression whose value is 4.

Furthermore, we can construct strings such as:

$$\texttt{"The sum of "} +\!\!+\ \mathit{show}\ x +\!\!+ \texttt{" and "} +\!\!+\ \mathit{show}\ y +\!\!+ \texttt{" is "} \\ \quad +\!\!+\mathit{show}\ (x + y) +\!\!+ \texttt{"."}$$

with no difficulty. In particular, because $(+\!\!+)$ is right associative, the number of steps to construct this string is directly proportional to its total length, and we can't expect to do any better than that. (Since $(+\!\!+)$ needs to reconstruct its left argument, if it were left associative the above expression would repeatedly reconstruct the same sub-string on each application of $(+\!\!+)$. If the total string length were n, then in the worst case the number of steps needed to do this would be proportional to n^2, instead of proportional to n in the case where $(+\!\!+)$ is right associative.)

Unfortunately, this strategy breaks down when construction of the list is nested. A particularly nasty version of this problem arises for tree-shaped data structures. Consider a function *showTree* that converts a value of type *Tree* into a string, as in:

$$\mathit{showTree}\ (\mathit{Branch}\ (\mathit{Branch}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 3))\ (\mathit{Leaf}\ 4)) \\ \Longrightarrow \texttt{"< <2|3>|4>"}$$

We can define this behavior straightforwardly as follows:

$$\begin{array}{l} \mathit{showTree} :: (\mathit{Show}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{String} \\ \mathit{showTree}\ (\mathit{Leaf}\ x) \\ \qquad\qquad = \mathit{show}\ x \\ \mathit{showTree}\ (\mathit{Branch}\ l\ r) \\ \qquad\qquad = \texttt{"<"} +\!\!+\ \mathit{showTree}\ l +\!\!+ \texttt{"|"} +\!\!+\ \mathit{showTree}\ r +\!\!+ \texttt{">"} \end{array}$$

Each of the recursive calls to *showTree* introduces more applications of (⧺), but since they are nested, a large amount of list reconstruction takes place (similar to the problem that would arise if (⧺) were left associative). If the tree being converted has size n, then in the worst case the number of steps needed to perform this conversion is proportional to n^2. This is no good!

To restore linear complexity, suppose we had a function *shows*:

```
shows :: (Show a) ⇒ a → String → String
```

which takes a showable value and a string and returns that string with the value's representation concatenated at the front. For example, we would expect *shows* (2 + 2) `"hello"` to return the string `"4hello"`. The string argument should be thought of as an "accumulator" for the final result.

Using *shows* we can define a more efficient version of *showTree* which, like *shows*, has a string accumulator argument. Let's call this function *showsTree*:

```
showsTree :: (Show a) ⇒ Tree a → String → String
showsTree (Leaf x) s
        = shows x s
showsTree (Branch l r) s
        = "<" ⧺ showsTree l ("|" ⧺ showsTree r (">" ⧺ s))
```

This function requires a number of steps directly proportional to the size of the tree, thus solving our efficiency problem. To see why this is so, note that the accumulator argument s is never reconstructed. It is simply passed as an argument in one recursive call to *shows* or *showsTree*, and is incrementally extended to its left using (⧺).

showTree can now be re-defined in terms of *showsTree* using an empty accumulator:

```
showTree t = showsTree t ""
```

Exercise B.1 Prove that this version of *showTree* is equivalent to the old.

Although this solves our efficiency problem, the presentation of this function (and others like it) can be improved somewhat. First, let's create a type synonym (part of the Standard Prelude):

```
type ShowS = String → String
```

Second, we can avoid carrying accumulators around, and also avoid amassing parentheses at the right end of long sequences of concatenations, by using functional composition:

```
showsTree :: (Show a) ⇒ Tree a → ShowS
showsTree (Leaf x)
      = shows x
showsTree (Branch l r)
      = ("<"++) ∘ showsTree l ∘ ("|"++) ∘ showsTree r ∘ (">"++)
```

> **Details:** This can be simplified slightly more by noting that ("c"++) is equivalent to ('c':) for any character *c*.

Something more important than just tidying up the code has come about by this transformation: We have raised the presentation from an *object level* (in this case, strings) to a *function level.* You can read the type signature of *showsTree* as saying that *showsTree* maps a tree into a *showing function.* Functions like ("<"++) and ("a string"++) are primitive showing functions, and we build up more complex ones by function composition.

The actual *Show* class in Haskell has two additional levels of complexity (and functionality): (1) the ability to specify the *precedence* of a string being generated, which is important when *show*ing a data type that has infix constructors, since it determines when parentheses are needed, and (2) a function for *show*ing a *list* of values of the type under consideration, since lists have special syntax in Haskell and are so commonly used that they deserve special treatment. The full definition of the *Show* class is given by:

```
class Show a where
    showsPrec :: Int → a → ShowS
    showList :: [a] → ShowS

    showList []
      = showString "[]"
    showList (x : xs)
      = showChar '[' ∘ shows x ∘ showl xs
        where showl [] = showChar ']'
              showl (x : xs) = showString ", " ∘ shows x ∘ showl xs
```

Note the default method for *showList*, and its "function level" style of definition.

In addition to this class declaration the Standard Prelude defines the following functions, which return us to where we started our journey in this section:

```
shows :: (Show a) ⇒ a → ShowS
```

$$
\begin{array}{l}
\mathit{shows} = \mathit{showsPrec}\ 0 \\
\\
\mathit{show} :: (\mathit{Show}\ a) \Rightarrow a \rightarrow \mathit{String} \\
\mathit{show}\ x = \mathit{shows}\ x\ \texttt{""}
\end{array}
$$

Some details about *showsPrec* can be found in the Haskell Report, but if you are not displaying constructors in infix notation, the precedence can be ignored. Furthermore, the default method for *showList* is perfectly good for most uses of lists that you will encounter. Thus, for example, we can finish our *Tree* example by declaring it to be an instance of the class *Show* very simply as:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Show}\ a) \Rightarrow \mathit{Show}\ (\mathit{Tree}\ a)\ \textbf{where} \\
\quad \mathit{showsPrec}\ n = \mathit{showsTree}
\end{array}
$$

B.5 The Read Class

Now that we can convert trees into strings, let's turn to the inverse problem: converting strings into trees. The basic idea is to define a *parser* for a type a, which at first glance seems as if it should be a function of type $\mathit{String} \rightarrow a$. This simple approach has two problems, however: (1) it's possible that the string is ambiguous, leading to more than one way to interpret it as a value of type a, and (2) it's possible that only a prefix of the string will parse correctly. Thus we choose instead to return a list of (a, String) pairs as the result of a parse. If all goes well we will always get a singleton list such as $[(v, \texttt{""})]$ as the result of a parse, but we cannot count on it (in fact, when recursively parsing sub-strings, we will expect a singleton list with a *non-empty* trailing string).

The Standard Prelude provides a type synonym for parsers of the kind just described:

$$\textbf{type}\ \mathit{ReadS}\ a = \mathit{String} \rightarrow [(a, \mathit{String})]$$

and also defines a function *reads* that by analogy is similar to *shows*:

$$\mathit{reads} :: (\mathit{Read}\ a) \Rightarrow \mathit{ReadS}\ a$$

We will return later to the precise definition of this function, but for now let's use it to define a parser for the *Tree* data type, whose string representation is as described in the previous section. List comprehensions give us a convenient idiom for constructing such parsers:[1]

[1] An even more elegant approach to parsing uses monads and parser combinators. These are part of a standard parsing library distributed with most Haskell systems.

$$
\begin{array}{l}
\mathit{readsTree} :: (\mathit{Read}\ a) \Rightarrow \mathit{ReadS}\ (\mathit{Tree}\ a) \\
\mathit{readsTree}\ (\texttt{'<'} : s) = [(\mathit{Branch}\ l\ r, u) \mid (l, \texttt{'|'} : t) \leftarrow \mathit{readsTree}\ s, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (r, \texttt{'>'} : u) \leftarrow \mathit{readsTree}\ t] \\
\mathit{readsTree}\ s = [(\mathit{Leaf}\ x, t) \mid (x, t) \leftarrow \mathit{reads}\ s]
\end{array}
$$

Let's take a moment to examine this function definition in detail. There are two main cases to consider: If the string has the form $\texttt{'<'} : s$ we should have the representation of a branch, in which case parsing s as a tree should yield a left branch l followed by a string of the form $\texttt{'|'} : t$; parsing t as a tree should then yield the right branch r followed by a string of the form $\texttt{'>'} : u$. The resulting tree *Branch* l r is then returned, along with the trailing string u. Note the expressive power we get from the combination of pattern matching and list comprehension.

If the initial string is not of the form $\texttt{'<'} : s$, then we must have a leaf, in which case the string is parsed using the generic *reads* function, and the result is directly returned.

If we accept on faith for the moment that there is a *Read* instance for *Int* that behaves as one would expect, e.g.:

$$
\begin{array}{l}
(\mathit{reads}\ \texttt{"5 golden rings"}) :: [(\mathit{Int}, \mathit{String})] \\
\Longrightarrow [(5, \texttt{" golden rings"})]
\end{array}
$$

then you should be able to verify the following calculations:

$$
\begin{array}{l}
\mathit{readsTree}\ \texttt{"< <1|2>|3>"} \\
\Longrightarrow
\end{array}
$$

There are a couple of shortcomings, however, in our definition of *readsTree*. One is that the parser is quite rigid in that it allows no "white space" (such as extra spaces, tabs, or line feeds) before or between the elements of the tree representation. The other is that the way we parse our punctuation symbols (`'<'`, `'|'`, and `'>'`) is quite different from the way we parse leaf values and sub-trees. This lack of uniformity makes the function definition harder to read.

We can address both of these problems by using a *lexical analyzer*, which parses a string into primitive "lexemes" defined by some rules about the string construction. The Standard Prelude defines a lexical analyzer:

$$
\mathit{lex} :: \mathit{ReadS}\ \mathit{String}
$$

whose lexical rules are those of the Haskell language, which can be found in the Haskell Report. For our purposes, an informal explanation is sufficient:

lex normally returns a singleton list containing a pair of strings: the first string is the first lexeme in the input string, and the second string is the remainder of the input. White space – including Haskell comments – is completely ignored. If the input string is empty or contains only white-space and comments, *lex* returns `[("","")]`; if the input is not empty in this sense, but also does not begin with a valid lexeme after any leading white-space, *lex* returns `[]`.

Using this lexical analyzer, our tree parser can be rewritten as:

```
readsTree :: (Read a) ⇒ ReadS (Tree a)
readsTree s = [(Branch l r, x) | ("<", t) ← lex s,
                                 (l, u) ← readsTree t,
                                 ("|", v) ← lex u,
                                 (r, w) ← readsTree v,
                                 (">", x) ← lex w]
              ⧺
              [(Leaf x, t) | (x, t) ← reads s]
```

This definition solves both problems mentioned earlier: white-space is suitably ignored, and parsing of sub-strings has a more uniform structure.

To tie all of this together, let's first look at the definition of the class *Read* in the Standard Prelude:

```
class Read a where
    readsPrec :: Int → ReadS a
    readList :: ReadS [a]

    readList = readParen False (λr → [pr | ("[", s) ← lex r,
                                           pr ← readl s])
                      where readl s = [([], t) | ("]", t) ← lex s]⧺
                                      [(x : xs, u) | (x, t) ← reads s,
                                                     (xs, u) ← readl′ t]
                            readl′ s = [([], t) | ("]", t) ← lex s]⧺
                                       [(x : xs, v) | (",", t) ← lex s,
                                                      (x, u) ← reads t,
                                                      (xs, v) ← readl′ u]

readParen :: Bool → ReadS a → ReadS a
readParen b g = if b then mandatory else optional
                where optional r = g r ⧺ mandatory r
                      mandatory r = [(x, u) | ("(", s) ← lex r,
                                        (x, t) ← optional s,
                                        (")", u) ← lex t]
```

The default method for *readList* is rather tedious, but otherwise straightforward.

reads can now be defined, along with an even higher-level function, *read*:

```
reads :: (Read a) ⇒ ReadS a
reads = readsPrec 0

read :: (Read a) ⇒ String → a
read s = case [x | (x, t) ← reads s, ("", "") ← lex t] of
                [x] → x
                [] → error "PreludeText.read: no parse"
                _ → error "PreludeText.read: ambiguous parse"
```

The definition of *reads* (like *shows*) should not be surprising. The definition of *read* assumes that exactly one parse is expected, and thus causes a run-time error if there is no unique parse or if the input contains anything more than a representation of exactly one value of type *a* (and possibly comments and white-space).

You can test that the *Read* and *Show* instances for a particular type are working correctly by applying (*read* ∘ *show*) to a value in that type, which in most situations should be the identity function.

B.6 The Index Class

The Standard Prelude defines a type class of array indices:

```
class (Ord a) ⇒ Ix a where
    range :: (a, a) → [a]
    index :: (a, a) → a → Int
    inRange :: (a, a) → a → Bool
```

Arrays are defined elsewhere, but the index class is useful for other things besides arrays, so I will describe it here.

Instance declarations are provided for *Int*, *Integer*, *Char*, *Bool*, and tuples of *Ix* types; in addition, instances may be automatically derived for enumerated and tuple types. You should think of the primitive types as vector indices, and tuple types as indices of multidimensional rectangular arrays. Note that the first argument of each of the operations of class *Ix* is a pair of indices; these are typically the *bounds* (first and last indices) of an array. For example, the bounds of a 10-element, zero-origin vector with *Int* indices would be $(0, 9)$, while a 100 by 100 1-origin matrix might have the bounds $((1, 1), (100, 100))$. (In many other languages, such bounds would be written in a form like $1 : 100, 1 : 100$, but the present form fits the type system better, since each bound is of the same type as a general index.)

The *range* operation takes a bounds pair and produces the list of indices lying between those bounds, in index order. For example,

$$\begin{aligned}&\textit{range}\ (0, 4) \Longrightarrow [0, 1, 2, 3, 4]\\&\textit{range}\ ((0, 0), (1, 2)) \Longrightarrow [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]\end{aligned}$$

The *inRange* predicate determines whether an index lies between a given pair of bounds. (For a tuple type, this test is performed componentwise, and then combined with $(\wedge)$.) Finally, the *index* operation determines the (zero-based) position of an index within a bounded range; for example:

$$\begin{aligned}&\textit{index}\ (1, 9)\ 2 \Longrightarrow 1\\&\textit{index}\ ((0, 0), (1, 2))\ (1, 1) \Longrightarrow 4\end{aligned}$$

B.7 The Numeric Classes

The *Num* class and the numeric class hierarchy were briefly described in Section 11.4. Figure B.1 gives the full class declarations.

```
class (Eq a, Show a) ⇒ Num a where
    (+), (−), (∗) :: a → a → a
    negate :: a → a
    abs, signum :: a → a
    fromInteger :: Integer → a

class (Num a, Ord a) ⇒ Real a where
    toRational :: a → Rational

class (Real a, Enum a) ⇒ Integral a where
    quot, rem, div, mod :: a → a → a
    quotRem, divMod :: a → a → (a, a)
    toInteger :: a → Integer

class (Num a) ⇒ Fractional a where
    (/) :: a → a → a
    recip :: a → a
    fromRational :: Rational → a

class (Fractional a) ⇒ Floating a where
    pi :: a
    exp, log, sqrt :: a → a
    (∗∗), logBase :: a → a → a
    sin, cos, tan :: a → a
    asin, acos, atan :: a → a
    sinh, cosh, tanh :: a → a
    asinh, acosh, atanh :: a → a

class (Real a, Fractional a) ⇒ RealFrac a where
    properFraction :: (Integral b) ⇒ a → (b, a)
    truncate, round :: (Integral b) ⇒ a → b
    ceiling, floor :: (Integral b) ⇒ a → b

class (RealFrac a, Floating a) ⇒ RealFloat a where
    floatRadix :: a → Integer
    floatDigits :: a → Int
    floatRange :: a → (Int, Int)
    decodeFloat :: a → (Integer, Int)
    encodeFloat :: Integer → Int → a
    exponent :: a → Int
    significand :: a → a
    scaleFloat :: Int → a → a
    isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE
                              :: a → Bool
```

Figure B.1: Standard Numeric Classes

Appendix C

Built-in Types Are Not Special

Throughout this text we have introduced many "built-in" types such as lists, tuples, integers, and characters. We have also shown how new user-defined types can be defined. Aside from special syntax, you might be wondering if the built-in types are in any way more special than the user-defined ones. The answer is *no.* The special syntax is for convenience and for consistency with historical convention, but has no semantic consequence.

We can emphasize this point by considering what the type declarations would look like for these built-in types if in fact we were allowed to use the special syntax in defining them. For example, the *Char* type might be written as:

```
data Char = 'a' | 'b' | 'c' | ...      -- This is not valid
          | 'A' | 'B' | 'C' | ...      -- Haskell code!
          | '1' | '2' | '3' | ...
```

These constructor names are not syntactically valid; to fix them we would have to write something like:

```
data Char = Ca | Cb | Cc | ...
          | CA | CB | CC | ...
          | C1 | C2 | C3 | ...
```

Even though these constructors are actually more concise, they are quite unconventional for representing characters, and thus the special syntax is used instead.

In any case, writing "pseudo-Haskell" code in this way helps us to see through the special syntax. We see now that *Char* is just a data type consisting of a large number of nullary (meaning they take no arguments) constructors. Thinking of *Char* in this way makes it clear why, for example, we can pattern-match against characters; i.e., we would expect to be able to do so for any of a data type's constructors.

Similarly, using pseudo-Haskell, we could define *Int* and *Integer* by:

```
                        -- more pseudo-code:
data Int = (−2ˆ29) | ... | −1 | 0 | 1 | ... | (2ˆ29 − 1)
data Integer = ... − 2 | −1 | 0 | 1 | 2...
```

(Recall that -2^{29} to 2^{29-1} is the minimum range for the *Int* data type.) *Int* is clearly a much larger enumeration than *Char*, but it's still finite! In contrast, the pseudo-code for *Integer* (the type of arbitrary precision integers) is intended to convey an *infinite* enumeration (and in that sense only, the *Integer* data type *is* somewhat special).

Haskell has a data type called *unit* which has exactly one value: (). The name of this data type is also written (). This is trivially expressed in Haskell pseudo-code:

```
data () = ()      -- more pseudo-code
```

Tuples are also easy to define playing this game:

```
data (a, b) = (a, b)      -- more pseudo-code
data (a, b, c) = (a, b, c)
data (a, b, c, d) = (a, b, c, d)
```

and so on. Each declaration above defines a tuple type of a particular length, with parentheses playing a role in both the expression syntax (as data constructor) and type-expression syntax (as type constructor). By "and so on" we mean that there are an infinite number of such declarations, reflecting the fact that tuples of all finite lengths are allowed in Haskell.

The list data type is also easily handled in pseudo-Haskell, and more interestingly, it is recursive:

```
data [a] = [ ] | a : [a]      -- more pseudo-code
infixr 5:
```

We can now see clearly what we described about lists earlier: [ ] is the empty list, and (:) is the infix list constructor; thus $[1, 2, 3]$ must be equivalent to the list $1 : 2 : 3 : [\,]$. (Note that (:) is right associative.) The type of [ ] is $[a]$, and the type of (:) is $a \to [a] \to [a]$.

> **Details:** The way (:) is defined here is actually legal syntax—infix constructors are permitted in **data** declarations, and are distinguished from infix operators (for pattern-matching purposes) by the fact that they must begin with a colon (a property trivially satisfied by ":").

At this point the reader should note carefully the differences between tuples and lists, which the above definitions make abundantly clear. In particular, note the recursive nature of the list type whose elements are homogeneous and of arbitrary length, and the non-recursive nature of a (particular) tuple type whose elements are heterogeneous and of fixed length. The typing rules for tuples and lists should now also be clear:

For $(e1, e2, ..., en)$, $n \geqslant 2$, if Ti is the type of ei, then the type of the tuple is $(T1, T2, ..., Tn)$.

For $[e1, e2, ..., en]$,$n \geqslant 0$, each ei must have the same type T, and the type of the list is $[T]$.

Appendix D

Pattern-Matching Details

In this chapter we will look at Haskell's pattern-matching process in greater detail.

Haskell defines a fixed set of patterns for use in case expressions and function definitions. Pattern matching is permitted using the constructors of any type, whether user-defined or pre-defined in Haskell. This includes tuples, strings, numbers, characters, etc. For example, here's a contrived function that matches against a tuple of "constants:"

$$\begin{array}{l} \mathit{contrived} :: ([\,a\,], \mathit{Char}, (\mathit{Int}, \mathit{Float}), \mathit{String}, \mathit{Bool}) \rightarrow \mathit{Bool} \\ \mathit{contrived}\ ([\,], \texttt{'b'}, (1, 2.0), \texttt{"hi"}, \mathit{True}) = \mathit{False} \end{array}$$

This example also demonstrates that *nesting* of patterns is permitted (to arbitrary depth).

Technically speaking, *formal parameters* to functions are also patterns—it's just that they *never fail to match a value.* As a "side effect" of a successful match, the formal parameter is bound to the value it is being matched against. For this reason patterns in any one equation are not allowed to have more than one occurrence of the same formal parameter.

A pattern that may fail to match is said to be *refutable*; for example, the empty list [ ] is refutable. Patterns such as formal parameters that never fail to match are said to be *irrefutable.* There are three other kinds of irrefutable patterns, which are summarized below.

As-Patterns Sometimes it is convenient to name a pattern for use on the right-hand side of an equation. For example, a function that duplicates the first element in a list might be written as:

$$f\ (x : xs) = x : x : xs$$

Note that $x : xs$ appears both as a pattern on the left-hand side, and as an expression on the right-hand side. To improve readability, we might prefer to write $x{:}xs$ just once, which we can achieve using an *as-pattern* as follows:[1]

$$f\ s@(x : xs) = x : s$$

Technically speaking, as-patterns always result in a successful match, although the sub-pattern (in this case $x : xs$) could, of course, fail.

Wildcards Another common situation is matching against a value we really care nothing about. For example, the functions *head* and *tail* can be written as:

$$\begin{aligned}&\mathit{head}\ (x : \_) = x\\ &\mathit{tail}\ (\_ : xs) = xs\end{aligned}$$

in which we have "advertised" the fact that we don't care what a certain part of the input is. Each wildcard will independently match anything, but in contrast to a formal parameter, each will bind nothing; for this reason more than one are allowed in an equation.

Lazy Patterns There is one other kind of pattern allowed in Haskell. It is called a *lazy pattern*, and has the form $\sim pat$. Lazy patterns are *irrefutable*: matching a value v against $\sim pat$ always succeeds, regardless of pat. Operationally speaking, if an identifier in pat is later "used" on the right-hand-side, it will be bound to that portion of the value that would result if v were to successfully match pat, and $\perp$ otherwise.

Lazy patterns are useful in contexts where infinite data structures are being defined recursively. For example, infinite lists are an excellent vehicle for writing *simulation* programs, and in this context the infinite lists are often called *streams*.

Pattern-Matching Semantics

So far we have discussed how individual patterns are matched, how some are refutable, some are irrefutable, etc. But what drives the overall process? In what order are the matches attempted? What if none succeed? This section addresses these questions.

[1] Another advantage to doing this is that a naive implementation might otherwise completely reconstruct $x : xs$ rather than re-use the value being matched against.

Pattern matching can either *fail*, *succeed* or *diverge*. A successful match binds the formal parameters in the pattern. Divergence occurs when a value needed by the pattern diverges (i.e. is non-terminating) or results in an error ($\perp$). The matching process itself occurs "top-down, left-to-right." Failure of a pattern anywhere in one equation results in failure of the whole equation, and the next equation is then tried. If all equations fail, the value of the function application is $\perp$, and results in a run-time error.

For example, if *bot* is a divergent or erroneous computation, and if $[1,2]$ is matched against $[0, \mathit{bot}]$, then 1 fails to match 0, so the result is a failed match. But if $[1,2]$ is matched against $[\mathit{bot}, 0]$, then matching 1 against *bot* causes divergence (i.e. $\perp$).

The only other twist to this set of rules is that top-level patterns may also have a boolean *guard*, as in this definition of a function that forms an abstract version of a number's sign:

$$\begin{aligned} &\mathit{sign}\ x \mid x > 0 = 1 \\ &\phantom{\mathit{sign}\ x\ } \mid x == 0 = 0 \\ &\phantom{\mathit{sign}\ x\ } \mid x < 0 = -1 \end{aligned}$$

Note here that a sequence of guards is given for a single pattern; as with patterns, these guards are evaluated top-down, and the first that evaluates to *True* results in a successful match.

An Example The pattern-matching rules can have subtle effects on the meaning of functions. For example, consider this definition of *take*:

$$\begin{aligned} &\mathit{take}\ 0\ \_ = [\,] \\ &\mathit{take}\ \_\ [\,] = [\,] \\ &\mathit{take}\ n\ (x : \mathit{xs}) = x : \mathit{take}\ (n-1)\ \mathit{xs} \end{aligned}$$

and this slightly different version (the first 2 equations have been reversed):

$$\begin{aligned} &\mathit{take1}\ \_\ [\,] = [\,] \\ &\mathit{take1}\ 0\ \_ = [\,] \\ &\mathit{take1}\ n\ (x : \mathit{xs}) = x : \mathit{take1}\ (n-1)\ \mathit{xs} \end{aligned}$$

Now note the following:

$$\begin{array}{lcl} \mathit{take}\ 0\ \mathit{bot} & \Longrightarrow & [\,] \\ \mathit{take1}\ 0\ \mathit{bot} & \Longrightarrow & \perp \\ \\ \mathit{take}\ \mathit{bot}\ [\,] & \Longrightarrow & \perp \\ \mathit{take1}\ \mathit{bot}\ [\,] & \Longrightarrow & [\,] \end{array}$$

We see that *take* is "more defined" with respect to its second argument, whereas *take1* is more defined with respect to its first. It is difficult to say in this case which definition is better. Just remember that in certain applications, it may make a difference. (The Standard Prelude includes a definition corresponding to *take*.)

Case Expressions

Pattern matching provides a way to "dispatch control" based on structural properties of a value. However, in many circumstances we don't wish to define a *function* every time we need to do this. Haskell's *case expression* provides a way to solve this problem. Indeed, the meaning of pattern matching in function definitions is specified in the Haskell Report in terms of case expressions, which are considered more primitive. In particular, a function definition of the form:

$$\begin{array}{l} f p_{11} ... p_{1k} = e_1 \\ ... \\ f p_{n1} ... p_{nk} = e_n \end{array}$$

where each p_{ij} is a pattern, is semantically equivalent to:

$$\begin{array}{ll} f\ x1\ x2\ ...\ xk = \textbf{case}\ (x1, ..., xk)\ \textbf{of} & (p_{11}, ..., p_{1k}) \rightarrow e_1 \\ & ... \\ & (p_{n1}, ..., p_{nk}) \rightarrow e_n \end{array}$$

where the *xi* are new identifiers. For example, the definition of *take* given earlier is equivalent to:

$$\begin{array}{l} take\ m\ ys = \textbf{case}\ (m, ys)\ \textbf{of} \\ \qquad\qquad (0, \_) \rightarrow [\,] \\ \qquad\qquad (\_, [\,]) \rightarrow [\,] \\ \qquad\qquad (n, x : xs) \rightarrow x : take\ (n-1)\ xs \end{array}$$

For type correctness, the types of the right-hand sides of a case expression or set of equations comprising a function definition must all be the same; more precisely, they must all share a common principal type.

The pattern-matching rules for case expressions are the same as we have given for function definitions.

Bibliography

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell (second edition).* Prentice Hall, London, 1998.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, New York, 1988.

[Chu41] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, 1941.

[Hin69] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[Hof79] D.R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid.* Vintage, New York, 1979.

[Hud89] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[Mil78] R.A. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, MA, 1990.

[Qui66] W.V.O. Quine. *The Ways of Paradox, and Other Essays.* Random House, New York, 1966.

[Sch24] M. Schönfinkel. Uber die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305, 1924.