The Haskell School of Music

— From Signals to Symphonies —

Paul Hudak

Yale University
Department of Computer Science

Version 2.6 (January 2014)

The Haskell School of Music
— From Signals to Symphonies —

Paul Hudak

Yale University
Department of Computer Science
New Haven, CT, USA

Version 2.6 (January 2014)

Cover image: *Euterpe*, the Greek Muse of Music
(attribution unknown)

Contents

List of Figures

List of Tables

Preface

In 2000 I wrote a book called *The Haskell School of Expression – Learning Functional Programming through Multimedia* [Hud00]. 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 [P^+03] is quite a bit different from conventional imperative or object-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 over twenty years ago, it has only recently begun to catch on in a significant way, 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 fifteen years ago, in an effort to combine work with play, I and my students wrote a Haskell library called *Haskore* for expressing high-level computer music concepts in a purely functional way [HMGW96, Hud96, Hud03]. Indeed, three of the chapters in *The Haskell School of Expression* summarize the basic ideas of this work. Soon after that, with the help of another student, I designed a Haskell library called *HasSound* that was, essentially, a Haskell interface to *csound* [Ver86] for doing sound synthesis and instrument design.

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 a rewrite of *The Haskell School of Expression* with a focus on computer music, based on, and greatly improving upon, the ideas in Haskore and HasSound.

The new Haskell library that incorporates all of this is called *Euterpea*.

Haskell was named after the logician Haskell B. Curry who, along with Alonzo Church, helped establish 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 programming paradigm). If you teach only Haskell language details to a C programmer, he or she is likely to write ugly, incomprehensible functional programs. But if you teach 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 also a form of expression, although Curry was more likely talking about writing and speaking. 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 computer music 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 not just to teach computer music concepts. Along the way you will also learn Haskell. There is no limit to what one might wish to do with computer 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—face the danger of being too limited in expressiveness.

You do not need to know much, if any, music theory to read this book, and you do not need to play an instrument. Of course, the more you know about music, the more you will be able to apply the concepts learned in this text in musically creative ways.

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 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 do not hide any details—I want Euterpea 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. 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. Those who succeed in this process often 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.

Finally, I note that some section titles are prefaced with the parenthetical phrase, “[**Advanced**]”. These sections may be skipped upon first reading, especially if the focus is on learning computer music concepts, as opposed to programming concepts.

Haskell Implementations

There are several implementations of Haskell, all available free on the Internet through the Haskell users’ website at `http://haskell.org`. However, the one that has dominated all others, and on which Euterpea is based, 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, various flavors of Unix, and Macs. The preferred way to install GHC is through the *Haskell Platform* (`http://hackage.haskell.org/platform/`). Any text editor can be used to create source files, but I prefer to use emacs (see `http://www.gnu.org/software/emacs`), along with its Haskell mode (see `http://projects.haskell.org/haskellmode-emacs/`). The entire Euterpea library, including the source code from this textbook, and installation instructions, can be found at `http://haskell.cs.yale.edu`.

Acknowledgements

I wish to thank my funding agencies—the National Science Foundation, the Defense Advanced Research Projects Agency, and Microsoft Research—for their generous support of research that contributed to the foundations of Euterpea. Yale University has provided me a stimulating and flexible environment to pursue my dreams for over thirty years, and I am especially thankful for its recent support of the Computing and the Arts initiative.

Tom Makucevich, a talented computer music practitioner and composer in New Haven, was the original motivator, and first user, of Haskore, which preceded Euterpea. Watching him toil endlessly with low-level csound programs was simply too much for me to bear! Several undergraduate students at Yale contributed to the original design and implementation of Haskore. I would like to thank in particular the contributions of Syam Gadde and Bo Whong, who co-authored the original paper on Haskore. Additionally, Matt Zamec helped me greatly in the creation of HasSound.

I wish to thank my more recent graduate students, in particular Hai (Paul) Liu, Eric Cheng, Donya Quick, and Daniel Winograd-Cort for their help in writing much of the code that constitutes the current Euterpea library. In addition, many students in my computer music classes at Yale provided valuable feedback through earlier drafts of the manuscript.

Finally, I wish to thank my wife, Cathy Van Dyke, my best friend and ardent supporter, whose love, patience, and understanding have helped me get through some bad times, and enjoy the good.

Happy Haskell Music Making!

Paul Hudak
New Haven
January 2012

Chapter 1

Overview of Computer Music, Euterpea, and Haskell

Computers are everywhere. And so is music! Although some might think of the two as being at best distant relatives, in fact they share many deep properties. Music comes from the soul, and is inspired by the heart, yet it has the mathematical rigor of computers. Computers have mathematical rigor of course, yet the most creative ideas in mathematics and computer science come from the soul, just like music. Both disciplines demand both left-brain and right-brain skills. It always surprises me how many computer scientists and mathematicians have a serious interest in music. It seems that those with a strong affinity or acuity in one of these disciplines is often strong in the other as well.

It is quite natural then to consider how the two might interact. In fact there is a long history of interactions between music and mathematics, dating back to the Greeks' construction of musical scales based on arithmetic relationships, and including many classical composers use of mathematical structures, the formal harmonic analysis of music, and many modern music composition techniques. Advanced music theory uses ideas from diverse branches of mathematics such as number theory, abstract algebra, topology, category theory, calculus, and so on.

There is also a long history of efforts to combine computers and music. Most consumer electronics today are digital, as are most forms of audio processing and recording. But in addition, digital musical instruments provide new modes of expression, notation software and sequencers have become standard tools for the working musician, and those with the most computer science savvy use computers to explore new modes of composition, transformation, performance, and analysis.

This textbook explores the fundamentals of computer music using a language-centric approach. In particular, the functional programming language *Haskell* is used to express all of the computer music concepts. Thus a by-product of learning computer music concepts will be learning how to program in Haskell. The core musical ideas are collected into a Haskell library called *Euterpea*. The name "Euterpea" is derived from *Euterpe*, who was one of the nine Greek muses, or goddesses of the arts, specifically the muse of music. A hypothetical picture of Euterpe graces the cover of this textbook.

1.1 The Note vs. Signal Dichotomy

The field of computer music has grown astronomically over the past several decades, and the material can be structured and organized along several dimensions. A dimension that proves particulary useful with respect to a programming language is one that separates *high-level* musical concerns from *low-level* musical concerns. Since a "high-level" programming language—namely Haskell—is used to program at both of these musical levels, to avoid confusion the terms *note level* and *signal level* will be used in the musical dimension.

At the *note level*, a *note* (i.e. pitch and duration) is the lowest musical entity that is considered, and everything else is built up from there. At this level, in addition to conventional representations of music, we can study interesting aspects of so-called *algorithmic composition*, including the use of fractals, grammar-based systems, stochastic processes, and so on. From this basis we can also study the harmonic and rhythmic *analysis* of music, although that is not currently an emphasis in this textbook. Haskell facilitates programming at this level through its powerful data abstraction facilities, higher-order functions, and declarative semantics.

In contrast, at the *signal level* the focus is on the actual sound generated in a computer music application, and thus a *signal* is the lowest entity that is considered. Sound is concretely represented in a digital computer by a discrete sampling of the continuous audio signal, at a high enough rate that human ears cannot distinguish the discrete from the continuous, usually 44,100 samples per second (the standard sampling rate used for CDs, mp3 files, and so on). But in Euterpea, these details are hidden: signals are treated abstractly as continuous quantities. This greatly eases the burden of programming with sequences of discrete values. At the signal level, we can study sound synthesis techniques (to simulate the sound of a conventional instrument, say, or something completely artificial), audio processing (e.g. determining the frequency spectrum of a signal), and special effects (reverb, panning, distortion, and so on).

Suppose for a moment that a musician is playing music using a metronome set at 96, which corresponds to 96 beats per minute. That means that one beat takes $60/96 = 0.625$ seconds. At a stereo sampling rate of 44,100 samples per second, that in turn translates into $2 \times 0.625 \times 44,100 = 55{,}125$ samples, and each sample typically occupies several bytes of computer memory. This is typical of the minimum memory requirements of a computation at the signal level. In contrast, at the note level, we only need some kind of operator or data structure that says "play this note," which requires a total of only a small handful of bytes. This dramatic difference highlights one of the key computational differences between programming at the note level versus the signal level.

Of course, many computer music applications involve both the note level *and* the signal level, and indeed there needs to be a mechanism to mediate between the two. Although such mediation can take many forms, it is for the most part straightforward. Which is another reason why the distinction between the note level and the signal level is so natural.

This textbook begins with a treatment of the note level (Chapters 1-17) and follows with a treatment of the signal level (Chapters 18-21). If you are interested only in the signal level, you could skip Chapters 8-17.

1.2 Basic Principles of Programming

Programming, in its broadest sense, is *problem solving*. It begins by recognizing problems that can and should be solved using a digital computer. Thus the first step in programming is answering the question, "What problem am I trying to solve?"

Once the problem is understood, a solution must be found. This may not be easy, of course, and in fact you may discover several solutions, so a way to measure success is needed. There are various dimensions in which to do this, including correctness ("Will I get the right answer?") and efficiency ("Will it run fast enough, or use too much memory?"). 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 in programming.

The last measure of success mentioned above—clarity of a program—is somewhat elusive: difficult to quantify and measure. Nevertheless, in large software systems clarity is an especially important goal, since such systems are worked on by many people over long periods of time, and evolve considerably as they mature. Having easy-to-understand code makes it much easier to modify.

In the area of computer music, there is another reason why clarity is important: namely, that the code often represents the author's thought process, musical intent, and artistic choices. A conventional musical score does not say much about what the composer thought as she wrote the music, but a program often does. So when you write your programs, write them for others to see, and aim for elegance and beauty, just like the musical result that you desire.

Programming is itself a creative process. Sometimes programming solutions (or artistic creations) come to mind all at once, with little effort. More often, however, they are discovered only after lots of hard work! We may write a program, modify it, throw it away and start over, give up, start again, and so on. It is important to realize that such hard work and reworking of programs is the norm, and in fact you are encouraged to get into the habit of doing so. Do not always be satisfied with your first solution, and always be prepared to go back and change or even throw away those parts of your program that you are not happy with.

1.3 Computation by Calculation

It is helpful when learning a new programming language to have a good grasp of how programs in that language are executed—in other words, an understanding of what a program *means*. The execution of Haskell programs is perhaps best understood as *computation by calculation*. Programs in Haskell can be viewed as *functions* whose input is that of the problem being solved, and whose output is the desired result—and the behavior of functions can be effectively understood as computation by calculation.

An example involving numbers might help to demonstrate these ideas.

Numbers are used in many applications, and computer music is no exception. For example, integers might be used to represent pitch, and floating-point numbers might be used to perform calculations involving frequency or amplitude.

Suppose we wish to perform an arithmetic calculation such as $3 \times (9+5)$. In Haskell this would be written as $3*(9+5)$, since most standard computer keyboards and text editors do not recognize the special symbol $\times$. The result can be calculated 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:[1]

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

Even though this calculation takes two extra steps, it at least gives the same, correct answer. Indeed, an important property of each and every program written in Haskell is that it will always yield the same answer when given the same inputs, regardless of the order chosen to perform the calculations.[2] 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 required fewer steps than the second, and thus it is said to be more *efficient.* Efficiency in both space (amount of memory used) and time (number of steps executed) is important when searching for solutions to problems. Of course, if the computation returns the wrong answer, efficiency is a moot point. In general it is best to search first for an elegant (and correct!) solution to a problem, and later refine it for better performance. This strategy is sometimes summarized as, "Get it right first!"

The above calculations are fairly trivial, but much more sophisticated computations will be introduced soon enough. For starters—and to intro-

[1] This assumes that multiplication distributes over addition in the number system being used, a point that will be returned to later in the text.

[2] This is true as long as a non-terminating sequence of calculations is not chosen, another issue that will be addressed later.

duce the idea of a Haskell function—the arithmetic operations performed in the previous example can be *generalized* by defining a function to perform them for any numbers x, y, and z:

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

This equation defines *simple* as a function of three *arguments*, x, y, and z. In mathematical notation this definition might be written differently, such as one of the following:

$$simple(x, y, z) = x \times (y + z)$$
$$simple(x, y, z) = x \cdot (y + z)$$
$$simple(x, y, z) = x(y + z)$$

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:

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

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.

simple 3 9 5 is said to *evaluate* to 42. To express the fact 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:

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

Note that the same notation is used for these symbolic steps as for concrete ones. In particular, the arrow in the notation reflects the direction of formal reasoning, and nothing more. In general, if $e_1 \Rightarrow e_2$, then it is also true that $e_2 \Rightarrow e_1$.

These symbolic steps are also referred to as 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 spell out the justification for each step, whether symbolic or concrete, a calculation can be annotated 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. Computer music applications often have some kind of a mathematical basis, and that mathematics must be reflected somewhere in our programs. But how do we know if we got it right? Proof by calculation is one way to connect the problem specification with the program solution.

More broadly speaking, as the world begins to rely more and more on computers to accomplish not just ordinary tasks such as writing term papers, sending email, and social networking, but also life-critical tasks such as controlling medical procedures and guiding spacecraft, then the correctness of programs 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, will at least give us 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 naïve at worst. How do we write to a file, play a sound, draw a picture, 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.

In many ways this first chapter is the most difficult, since it contains the highest density of new concepts. If the reader has trouble with some of the concepts in this overview chapter, keep in mind that most of them will be revisited in later chapters. And do not hesitate to return to this chapter later to re-read difficult sections; they will likely be much easier to grasp at that time.

> **Details:** In the remainder of this textbook 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 circumstances the explanations will be offset in a shaded box such as this one, proceeded with the word "Details."

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

$$simple\ (simple\ 2\ 3\ 4)\ 5\ 6$$

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

1.4 Expressions and Values

In Haskell, the entities on which calculations are performed 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—every value is an expression, but not the other way around.

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 Haskell. The next chapter introduces examples of *constructor* atomic values, such as the musical notes C, D, Ef, Fs, etc., which in standard music notation are written C, D, E♭, F♯, etc., and are pronounced C, D, E-flat, F-sharp, etc. (In music theory, note names are called *pitch classes*.).

In addition, there are *structured* expressions (i.e., made from smaller pieces) such as the *list* of pitches $[C, D, Ef]$, the character/number *pair* (`'b'`, 4) (lists and pairs are different in a subtle way, to be described later), and the string `"Euterpea"`. Each of these structured expressions is also a value, since by themselves there is no further 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. As a final example, as was expained earlier, the expression *simple* 3 9 5 evaluates to the value 42.

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

$$x = x + 1$$

then here is what happens when trying to calculate the value of x:

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

Similarly, if a function f is defined as:

$$f\ x = f\ (x - 1)$$

then an expression such as f 42 runs into a similar problem:

$$\begin{array}{l} f\ 42 \\ \Rightarrow f\ 41 \\ \Rightarrow f\ 40 \\ \Rightarrow f\ 39 \\ \ldots \end{array}$$

Both of these clearly result in a never-ending sequence of calculations. Such expressions are said to not terminate, or *diverge*. In such cases the symbol $\perp$, pronounced "bottom," is used to denote the value of the expression. This means that every diverging computation in Haskell denotes the same

$\perp$ value,[3] reflecting the fact that, from an observer's point of view, there is nothing to distinguish one diverging computation from another.

1.5 Types

Every expression (and therefore every value) also has an associated *type*. It is helpful to think of types as sets of expressions (or values), in which members of the same set have much in common. Examples include the primitive atomic types *Integer* (the set of all integers) and *Char* (the set of all characters), the user-defined atomic type *PitchClass* (the set of all pitch classes, i.e. note names), as well as the structured types [*Integer*] and [*PitchClass*] (the sets of all lists of integers and lists of pitch classes, respectively), and *String* (the set of all Haskell strings).

The association of an expression or value with its type is very useful, and there is a special way of expressing it in Haskell. Using the examples of values and types above:

```
D           :: PitchClass
42          :: Integer
'a'         :: Char
"Euterpea"  :: String
[C, D, Ef]  :: [PitchClass]
('b', 4)    :: (Char, Integer)
```

Each association of an expression with its type is called a *type signature*.

> **Details:** Note that the names of specific types are capitalized, such as *Integer* and *Char*, as are the names of some atomic values such as *D* and *Fs*. These will never be confused in context, since things to the right of "::" are types, and things to the left are values. Note also that user-defined names of values are *not* capitalized, 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.

[3] Technically, each type has its own version of $\perp$.

Details: Literal characters are written enclosed in single forward quotes (apostrophes), as in `'a'`, `'A'`, `'b'`, `','`, `'!'`, `' '` (a space), and so on. (There are some exceptions, however; see the Haskell Report for details.) Strings are written enclosed in double quote characters, as in `"Euterpea"` above. The connection between characters and strings will be explained in a later chapter.

The "::" should be read "has type," as in "42 has type *Integer*." Note that square braces are used both to construct a list value (the left-hand side of (::) above), and to describe its type (the right-hand side above). Analogously, the round braces used for pairs are used in the same way. But also note that all of the elements in a list, however long, must have the same type, whereas the elements of a pair can have different 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 type errors.

1.6 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—T_1, say—as input, and returns values of (possibly) some other type—T_2, say—as output. In Haskell this is written $T_1 \rightarrow T_2$, and such a function is said to "map values of type T_1 to values of type T_2."[4] If there is more than one argument, the notation is extended with more arrows. For example, if the intent is that the function *simple* defined in the previous section has type *Integer* $\rightarrow$ *Integer* $\rightarrow$ *Integer* $\rightarrow$ *Integer*, we can include a type signature with the definition of *simple*:

```
simple       :: Integer → Integer → Integer → Integer
simple x y z = x * (y + z)
```

[4] In mathematics T_1 is called the *domain* of the function, and T_2 the *range*.

Details: When writing Haskell programs using a typical text editor, there typically will not be nice fonts and arrows as in $Integer \rightarrow Integer$. 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 the type system is said to *infer* the correct types.[5] Nevertheless, judicious placement of type signatures, as was done for *simple*, is a good habit, since type signatures are an effective form of documentation and help bring programming errors to light. In fact, it is a good habit to first write down the type of each function you are planning to define, as a first approximation to its full specification—a way to grasp its overall functionality before delving into its details.

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. 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 $f\ x\ y$).

Details: 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 ^! 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 infix operator as a value, it is enclosed in parentheses, such as when declaring its type, as in:

$$(+) :: Integer \rightarrow Integer \rightarrow Integer$$

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.

[5]There are a few exceptions to 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.

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

$[A, B, C]$
$[D, 42]$
$(-42, \mathit{Ef})$
$[(\texttt{'a'}, 3), (\texttt{'b'}, 5)]$
$\mathit{simple}\ \texttt{'a'}\ \texttt{'b'}\ \texttt{'c'}$
$(\mathit{simple}\ 1\ 2\ 3, \mathit{simple})$
$[\texttt{"I"}, \texttt{"love"}, \texttt{"Euterpea"}]$

For those expressions that are ill-typed, explain why.

1.7 Abstraction, Abstraction, Abstraction

The title of this section is the answer to the question: "What are the three most important ideas in programming?" Webster defines the verb "abstract" as follows:

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

In programming this happens when a pattern repeats itself and we wish to "separate" that pattern from the "particular instances" in which it appears. In this textbook this process is called the *abstraction principle*. The following sections introduce several different kinds of abstraction, using examples involving both simple numbers and arithmetic (things everyone should be familiar with) as well as musical examples (that are specific to Euterpea).

1.7.1 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 do not have the luxury of using greek letters on standard computer keyboards and/or text editors. So in Haskell we write:

$pi :: Double$
$pi = 3.141592653589793$

to associate the name *pi* with the number 3.141592653589793. The type signature in the first line declares *pi* to be a *double-precision floating-point number*, which mathematically and in Haskell is distinct from an integer.[6] Now the name *pi* can be used in expressions whenever it is in scope; it is an abstract representation, if you will, of the number 3.141592653589793. Furthermore, if there is ever a need to change a named value (which hopefully will not 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.

For a simple musical example, note first that in music theory, a *pitch* consists of a *pitch class* and an *octave*. For example, in Euterpea we simply write $(A, 4)$ to represent the pitch class A in the fourth octave. This particular note is called "concert A" (because it is often used as the note to which an orchestra tunes its instruments) or "A440" (because its frequency is 440 cycles per second). Because this particular pitch is so common, it may be desirable to give it a name, which is easily done in Haskell, as was done above for π:

```
concertA, a440 :: (PitchClass, Octave)
concertA = (A, 4)   -- concert A
a440     = (A, 4)   -- A440
```

Details: This example demonstrates the use of program *comments*. Any text to the right of "--" till the end of the line is considered to be a programmer 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, including across multiple lines.

This example demonstrates the (perhaps obvious) fact that several different names can be given to the same value—just as your brother John might have the nickname "Moose." Also note that the name *concertA* requires more typing than $(A, 4)$; nevertheless, it has more mnemonic value, and, if mistyped, will more likely result in a syntax error. For example, if you type "*concrtA*" by mistake, you will likely get an error saying, "Undefined

[6] We will have more to say about floating-point numbers later.

variable," whereas if you type "$(A, 5)$" you will not.

> **Details:** This example also demonstrates that two names having the same type can be combined into the same type signature, separated by a comma. Note finally, as a reminder, that these are names of values, and thus they both begin with a lowercase letter.

Consider now a problem whose solution requires writing some larger expression more than once. For example:

$$
\begin{aligned}
&x :: \mathit{Float} \\
&x = f\ (\mathit{pi} * r ** 2) + g\ (\mathit{pi} * r ** 2)
\end{aligned}
$$

> **Details:** $(**)$ is Haskell's floating-point exponentiation operator. Thus $\mathit{pi} * r ** 2$ is analogous to πr^2 in mathematics. $(**)$ has higher precedence than $(*)$ and the other binary arithmetic operators in Haskell.

Note in the definition of x that the expression $\mathit{pi} * r ** 2$ (presumably representing the area of a circle whose radius is r) is repeated—it has two instances—and thus, applying the abstraction principle, it can be separated from these instances. From the previous examples, doing this is straightforward—it is called *naming*—so we might choose to rewrite the single equation above as two:

$$
\begin{aligned}
&\mathit{area} = \mathit{pi} * r ** 2 \\
&x \qquad = f\ \mathit{area} + g\ \mathit{area}
\end{aligned}
$$

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

$$
\begin{aligned}
x = &\ \textbf{let}\ \mathit{area} = \mathit{pi} * r ** 2 \\
&\ \textbf{in}\ f\ \mathit{area} + g\ \mathit{area}
\end{aligned}
$$

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 were to write:

$$\begin{aligned}
&area = 42\\
&x \quad = \textbf{let}\ area = pi * r ** 2\\
&\qquad\quad \textbf{in}\ f\ area + g\ area
\end{aligned}$$

then there is no conflict of names—the "outer" *area* is completely different from the "inner" one enclosed in the **let** expression. Think of the inner *area* 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."

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.

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:

$$\begin{aligned}
&a = 42\\
&a = 43
\end{aligned}$$

nor is this:

$$\begin{aligned}
&a = 42\\
&b = 43\\
&a = 44
\end{aligned}$$

1.7.2 Functional Abstraction

The design of functions such as *simple* can be viewed as the abstraction principle in action. To see this using the example above involving the area of a circle, suppose the original program looked like this:

$$\begin{aligned}
&x :: \mathit{Float}\\
&x = f\ (pi * r_1 ** 2) + g\ (pi * r_2 ** 2)
\end{aligned}$$

Note that there are now two areas involved—one of a circle whose radius is r_1, the other r_2. Now the expressions in parentheses have a *repeating pattern of operations*. In discerning the nature of a repeating pattern it is sometimes helpful to first identify those things that are *not* repeating, i.e. those things that are *changing*. In the case above, it is the radius that is changing. A repeating pattern of operations can be abstracted as a *function* that takes the changing values as arguments. Using the function name *areaF* (for "area function") we can write:

$$\begin{aligned}&x = \textbf{let}\ areaF\ r = pi * r ** 2\\&\qquad \textbf{in}\ f\ (areaF\ r_1) + g\ (areaF\ r_2)\end{aligned}$$

This is a simple generalization of the previous example, where the function now takes the "variable quantity"—in this case the radius—as an argument. A very simple proof by calculation, in which *areaF* is unfolded where it is used, can be given to demonstrate that this program is equivalent to the old.

This application of the abstraction principle is called *functional abstraction*, since a sequence of operations is abstracted as a *function* such as *areaF*.

For a musical example, a few more concepts from Euterpea are first introduced, concepts that are addressed more formally in the next chapter:

1. In music theory a *note* is a *pitch* combined with a *duration*. Duration is measured in beats, and in Euterpea has type *Dur*. A note whose duration is one beat is called a whole note, one with duration 1/2 is called a half note, and so on. A note in Euterpea is the smallest entity, besides a rest, that is actually a performable piece of music, and its type is *Music Pitch* (other variations of this type will be introduced in later chapters).

2. In Euterpea there are functions:

$$\begin{aligned}&note :: Dur \rightarrow Pitch \rightarrow Music\ Pitch\\&rest\ \ :: Dur \rightarrow Music\ Pitch\end{aligned}$$

   such that $note\ d\ p$ is a note whose duration is d and pitch is p, and $rest\ d$ is a rest with duration d. For example, $note\ (1/4)\ (A, 4)$ is a quarter note concert A.

3. In Euterpea the following infix operators combine smaller *Music* values into larger ones:

$$\begin{aligned}&(:+:) :: Music\ Pitch \rightarrow Music\ Pitch \rightarrow Music\ Pitch\\&(:=:) :: Music\ Pitch \rightarrow Music\ Pitch \rightarrow Music\ Pitch\end{aligned}$$

   Intuitively:

   - m_1 :+: m_2 is the music value that represents the playing of m_1 followed by m_2.
   - m_1 :=: m_2 is the music value that represents the playing of m_1 and m_2 simultaneously.

4. Eutperepa also has a function $trans :: Int \rightarrow Pitch \rightarrow Pitch$ such that $trans\ i\ p$ is a pitch that is i semitones (half steps, or steps on a piano) higher than p.

Now for the example. Consider the simple melody:

$$note\ qn\ p_1 :+: note\ qn\ p_2 :+: note\ qn\ p_3$$

where qn is a quarter note:

$$qn = 1/4$$

Suppose we wish to harmonize each note with a note played a minor third lower. In music theory, a minor third corresponds to three semitones, and thus the harmonized melody can be written as:

$$\begin{aligned} mel = {} & (note\ qn\ p_1 :=: note\ qn\ (trans\ (-3)\ p_1))\ :+: \\ & (note\ qn\ p_2 :=: note\ qn\ (trans\ (-3)\ p_2))\ :+: \\ & (note\ qn\ p_3 :=: note\ qn\ (trans\ (-3)\ p_3)) \end{aligned}$$

Note as in the previous example a repeating pattern of operations—namely, the operations that harmonize a single note with a note three semitones below it. As before, to abstract a sequence of operations such as this, a function can be defined that takes the "variable quantities"—in this case the pitch—as arguments. We can take this one step further, however, by noting that in some other context we might wish to vary the duration. Recognizing this is to *anticipate* the need for abstraction. Calling this function *hNote* (for "harmonize note") we can then write:

$$\begin{aligned} & hNote && :: Dur \rightarrow Pitch \rightarrow Music\ Pitch \\ & hNote\ d\ p && = note\ d\ p :=: note\ d\ (trans\ (-3)\ p) \end{aligned}$$

There are three instances of the pattern in *mel*, each of which can be replaced with an application of *hNote*. This leads to:

$$\begin{aligned} & mel :: Music\ Pitch \\ & mel = hNote\ qn\ p_1 :+: hNote\ qn\ p_2 :+: hNote\ qn\ p_3 \end{aligned}$$

Again using the idea of unfolding described earlier in this chapter, it is easy to prove that this definition is equivalent to the previous one.

As with *areaF*, this use of *hNote* is an example of functional abstraction. In a sense, functional abstraction can be seen as a generalization of naming. That is, $area\ r_1$ is just a name for $pi * r_1 ** 2$, $hNote\ d\ p$ is just a name for $note\ d\ p :=: note\ d\ (trans\ (-3)\ p)$, and so on. Stated another way, named quantities such as *area*, *pi*, *concertA*, and *a440* defined earlier can be thought of as functions with no arguments.

Of course, the definition of *hNote* could also be hidden within *mel* using a **let** expression as was done in the previous example:

$$
\begin{aligned}
&mel :: Music\ Pitch \\
&mel = \mathbf{let}\ hNote\ d\ p = note\ d\ p :=: note\ d\ (trans\ (-3)\ p) \\
&\qquad\quad \mathbf{in}\ hNote\ qn\ p_1 :+: hNote\ qn\ p_2 :+: hNote\ qn\ p_3
\end{aligned}
$$

1.7.3 Data Abstraction

The value of *mel* is the sequential composition of three harmonized notes. But what if in another situation we must compose together five harmonized notes, or in other situations even more? In situations where the number of values is uncertain, it is useful to represent them in a *data structure*. For the example at hand, a good choice of data structure is a *list*, briefly introduced earlier, that can have any length. The use of a data structure motivated by the abstraction principle is one form of *data abstraction*.

Imagine now an entire list of pitches, whose length is not known at the time the program is written. What now? It seems that a function is needed to convert a list of pitches into a sequential composition of harmonized notes. Before defining such a function, however, there is a bit more to say about lists.

Earlier the example $[C, D, Ef]$ was given, a list of pitch classes whose type is thus $[PitchClass]$. A list with *no* elements is—not surprisingly—written $[\,]$, and is called the *empty list*.

To add a single element x to the front of a list xs, we write $x : xs$ in Haskell. (Note the naming convention used here; xs is the plural of x, and should be read that way.) For example, $C : [D, Ef]$ is the same as $[C, D, Ef]$. In fact, this list is equivalent to $C : (D : (Ef : [\,]))$, which can also be written $C : D : Ef : [\,]$ 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 is usually the case that the operator is *mathematically* associative, meaning that it does not matter which interpretation is chosen. If the interpretation *does* matter, mathematicians 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 many programming languages, including Haskell, each operator is defined to have a particular precedence level and to be left associative, right associative, or to have no associativity at all. 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 associativity can be overridden by using parentheses: thus $(a : b) : c$ is a valid Haskell expression (assuming that it is well-typed; it must be a list of lists), and is very different from $a : b : c$. A way to specify the precedence and associativity of user-defined operators will be discussed in a later chapter.

Returning now to the problem of defining a function (call it *hList*) to turn a list of pitches into a sequential composition of harmonized notes, we should first express what its type should be:

$$hList :: Dur \rightarrow [Pitch] \rightarrow Music\ Pitch$$

To define its proper behavior, it is helpful to consider, one by one, all possible cases that could arise on the input. First off, the list could be empty, in which case the sequential composition should be a *Music Pitch* value that has zero duration. So:

$$hList\ d\ [\,] = rest\ 0$$

The other possibility is that the list *is not* empty—i.e. it contains at least one element, say p, followed by the rest of the elements, say ps. In this case the result should be the harmonization of p followed by the sequential composition of the harmonization of ps. Thus:

$$hList\ d\ (p : ps) = hNote\ d\ p :+: hList\ d\ ps$$

Note that this part of the definition of *hList* is *recursive*—it refers to itself! But the original problem—the harmonization of $p : ps$—has been reduced to the harmonization of p (previously captured in the function *hNote*) and the harmonization of ps (a slightly smaller problem than the original one).

Combining these two equations with the type signature yields the complete definition of the function *hList*:

$$
\begin{aligned}
&\mathit{hList} &&:: \mathit{Dur} \to [\mathit{Pitch}] \to \mathit{Music}\ \mathit{Pitch}\\
&\mathit{hList}\ d\ [\,] &&= \mathit{rest}\ 0\\
&\mathit{hList}\ d\ (p : \mathit{ps}) &&= \mathit{hNote}\ d\ p \mathbin{:+:} \mathit{hList}\ d\ \mathit{ps}
\end{aligned}
$$

Recursion is a powerful technique that will be used many times in this textbook. It is also an example of a general problem-solving technique where a large problem is broken down into several smaller but similar problems; solving these smaller problems one-by-one leads to a solution to the larger problem.

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 $p : ps$ will successfully match any list with at least one element, while naming the first element p and the rest of the list ps). 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.

Given this definition of *hList* the definition of *mel* can be rewritten as:

$$\mathit{mel} = \mathit{hList}\ \mathit{qn}\ [p_1, p_2, p_3]$$

We can prove that this definition is equivalent to the old via calculation:

$$
\begin{aligned}
&\mathit{mel} = \mathit{hList}\ \mathit{qn}\ [p_1, p_2, p_3]\\
&\Rightarrow \mathit{hList}\ \mathit{qn}\ (p_1 : p_2 : p_3 : [\,])\\
&\Rightarrow \mathit{hNote}\ \mathit{qn}\ p_1 \mathbin{:+:} \mathit{hList}\ \mathit{qn}\ (p_2 : p_3 : [\,])\\
&\Rightarrow \mathit{hNote}\ \mathit{qn}\ p_1 \mathbin{:+:} \mathit{hNote}\ \mathit{qn}\ p_2 \mathbin{:+:} \mathit{hList}\ \mathit{qn}\ (p_3 : [\,])\\
&\Rightarrow \mathit{hNote}\ \mathit{qn}\ p_1 \mathbin{:+:} \mathit{hNote}\ \mathit{qn}\ p_2 \mathbin{:+:} \mathit{hNote}\ \mathit{qn}\ p_3 \mathbin{:+:} \mathit{hList}\ \mathit{qn}\ [\,]\\
&\Rightarrow \mathit{hNote}\ \mathit{qn}\ p_1 \mathbin{:+:} \mathit{hNote}\ \mathit{qn}\ p_2 \mathbin{:+:} \mathit{hNote}\ \mathit{qn}\ p_3 \mathbin{:+:} \mathit{rest}\ 0
\end{aligned}
$$

The first step above is not really a calculation, but rather a rewriting of the list syntax. The remaining calculations each represent an unfolding of *hList*.

Lists are perhaps the most commonly used data structure in Haskell, and there is a rich library of functions that operate on them. In subsequent chapters lists will be used in a variety of interesting computer music applications.

Exercise 1.4 Modify the definitions of *hNote* and *hList* so that they each take an extra argument that specifies the interval of harmonization (rather than being fixed at -3). Rewrite the definition of *mel* to take these changes into account.

1.8 Haskell Equality vs. Euterpean Equality

The astute reader will have objected to the proof just completed, arguing that the original version of *mel*:

$$hNote\ qn\ p_1 :+: hNote\ qn\ p_2 :+: hNote\ qn\ p_3$$

is not the same as the terminus of the above proof:

$$hNote\ qn\ p_1 :+: hNote\ qn\ p_2 :+: hNote\ qn\ p_3 :+: rest\ 0$$

Indeed, that reader would be right! As Haskell values, these expressions are *not* equal, and if you printed each of them you would get different results. So what happened? Did proof by calculation fail?

No, proof by calculation did not fail, since, as just pointed out, as Haskell values these two expressions are not the same, and proof by calculation is based on the equality of Haskell values. The problem is that a "deeper" notion of equivalence is needed, one based on the notion of *musical* equality. Adding a rest of zero duration to the beginning or end of any piece of music should not change what we *hear*, and therefore it seems that the above two expressions are *musically* equivalent. But it is unreasonable to expect Haskell to figure this out for the programmer!

As an analogy, consider the use of an ordered list to represent a set (which is unordered). The Haskell values $[x_1, x_2]$ and $[x_2, x_1]$ are not equal, yet in a program that "interprets" them as sets, they *are* equal.

The way this problem is approached in Euterpea is to formally define a notion of *musical interpretation*, from which the notion of *musical equivalence* is defined. This leads to a kind of "algebra of music" that includes, among others, the following axiom:

Figure 1.1: Polyphonic vs. Contrapuntal Interpretation

$$m \mathbin{:+:} \mathit{rest}\ 0 \equiv m$$

The operator ($\equiv$) should be read, "is musically equivalent to." With this axiom it is easy to see that the original two expressions above *are* in fact musically equivalent.

For a more extreme example of this idea, and to entice the reader to learn more about musical equivalence in later chapters, note that *mel*, given pitches $p_1 = \mathit{Ef}$, $p_2 = F$, $p_3 = G$, and duration $d = 1/4$, generates the harmonized melody shown in Figure 1.1; we can write this concretely in Euterpea as:

$$\begin{aligned} \mathit{mel}_1 = {} & (\mathit{note}\ (1/4)\ (\mathit{Ef}, 4) \mathbin{:=:} \mathit{note}\ (1/4)\ (C, 4)) \ \mathbin{:+:} \\ & (\mathit{note}\ (1/4)\ (F,\ 4) \mathbin{:=:} \mathit{note}\ (1/4)\ (D, 4)) \ \mathbin{:+:} \\ & (\mathit{note}\ (1/4)\ (G,\ 4) \mathbin{:=:} \mathit{note}\ (1/4)\ (E, 4)) \end{aligned}$$

The definition of mel_1 can then be seen as a *polyphonic* interpretation of the musical phrase in Figure 1.1, where each pair of notes is seen as a harmonic unit. In contrast, a *contrapuntal* interpretation sees two independent *lines*, or *voices*, in this case the line $\langle$E$\flat$,F,G$\rangle$ and the line $\langle$C,D,E$\rangle$. In Euterpea we can write this as:

$$\begin{aligned} \mathit{mel}_2 = {} & (\mathit{note}\ (1/4)\ (\mathit{Ef}, 4) \mathbin{:+:} \mathit{note}\ (1/4)\ (F, 4) \mathbin{:+:} \mathit{note}\ (1/4)\ (G, 4)) \\ & \ \mathbin{:=:} \\ & (\mathit{note}\ (1/4)\ (C,\ 4) \mathbin{:+:} \mathit{note}\ (1/4)\ (D, 4) \mathbin{:+:} \mathit{note}\ (1/4)\ (E, 4)) \end{aligned}$$

mel_1 and mel_2 are clearly not equal as Haskell values. Yet if they are played, they will *sound* the same—they are, in the sense described earlier, *musically equivalent*. But proving these two phrases musically equivalent will require far more than a simple axiom involving *rest* 0. In fact this can be done in an elegant way, using the algebra of music developed in Chapter 11.

1.9 Code Reuse and Modularity

There does not seem to be much repetition in the last definition of *hList*, so perhaps the end of the abstraction process has been reached. In fact, it is worth considering how much progress has been made. The original definition:

$$\begin{aligned} mel = {} & (note\ qn\ p_1 :=: note\ qn\ (trans\ (-3)\ p_1))\ :+: \\ & (note\ qn\ p_2 :=: note\ qn\ (trans\ (-3)\ p_2))\ :+: \\ & (note\ qn\ p_3 :=: note\ qn\ (trans\ (-3)\ p_3)) \end{aligned}$$

was replaced with:

$$mel = hList\ qn\ [p_1, p_2, p_3]$$

But additionally, definitions for the auxiliary functions *hNote* and *hList* were introduced:

$$\begin{aligned} & hNote && :: Dur \to Pitch \to Music\ Pitch \\ & hNote\ d\ p && = note\ d\ p :=: note\ d\ (trans\ (-3)\ p) \end{aligned}$$

$$\begin{aligned} & hList && :: Dur \to [Pitch] \to Music\ Pitch \\ & hList\ d\ [\,] && = rest\ 0 \\ & hList\ d\ (p : ps) && = hNote\ d\ p :+: hList\ d\ ps \end{aligned}$$

In terms of code size, the final program is actually larger than the original! So has the program improved in any way?

Things have certainly gotten better from the standpoint of "removing repeating patterns," and we could argue that the resulting program therefore is easier to understand. But there is more. Now that auxiliary functions such as *hNote* and *hList* have been defined, 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 building blocks, that can form the foundation of many applications.[7] In a later chapter, techniques will be introduced—most notably, *higher-order functions* and *polymorphism*—for improving the modularity of this example even more, and substantially increasing the ability to reuse code.

1.10 [Advanced] Programming with Numbers

In computer music programming, it is often necessary to program with numbers. For example, it is often convenient to represent pitch on a simple absolute scale using integer values. And when computing with analog signals that represent a particular sound wave, it is necessary to use floating point numbers as an approximation to the reals. So it is a good idea to understand precisely how numbers are represented inside a computer, and within a particular language such as Haskell.

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

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, and so on.

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. Clearly, for example, an irrational number such as π cannot be represented exactly; the best we can do is approximate it, or possibly write a program that computes it to whatever (finite) precision is needed 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 many 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*). Even worse, the behavior of arithmetic operations on floating-point numbers can vary somewhat depending on what kind of computer 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 $pi * (3 + 4) :: Float$ and $pi * 3 + pi * 4 :: Float$ 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:

$$\begin{array}{ll} 5 * (-0.123456 + 0.123457) & :: Float \Rightarrow 4.991889\text{e} - 6 \\ 5 * (-0.123456) + 5 * (0.123457) & :: Float \Rightarrow 5.00679\text{e} - 6 \end{array}$$

Although the discrepancy here is small, its very existence is worrisome, and in certain situations it could be disastrous. The precise behavior of floating-point numbers will not be discussed further in this textbook. 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). Although the representation of an *Integer* in Haskell is not normally something to be 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:

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

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 or 64 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 (and most other languages) provides another integer data type called *Int* that has maximum and minimum values that depend on the word-size of the particular computer being used. In other words, every value of type *Int* fits into one word of memory, and the primitive machine instructions for binary numbers can be used to manipulate them efficiently.[8] Unfortunately, this means that *overflow* or *underflow* er-

[8] The Haskell Report requires that every implementation support *Int*s at least in the

rors could occur when an *Int* value exceeds either the maximum or minimum values. Sadly, most implementations of Haskell (as well as most other languages) do not tell you when this happens. For example, in GHC running on a 32-bit processor, the following *Int* value:

$$
\begin{aligned}
& i :: \mathit{Int} \\
& i = 1234567890
\end{aligned}
$$

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,[9] 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 implied 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 textbook the numeric data types *Integer*, *Int*, *Float*, *Double*, *Rational*, and *Complex* will be used for a variety of different applications; for a discussion of the other number types, consult the Haskell Report. As these data types are used, there will be little discussion about their properties—this is not, after all, a book on numerical analysis—but a warning will be cast whenever reasoning about, for example, floating-point numbers, in a way that might not be technically sound.

range -2^{29} to $2^{29}-1$, inclusive. The GHC implementation running on a 32-bit processor, for example, supports the range -2^{31} to $2^{31}-1$.

[9] Actually, these bits 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 Euterpea.Music.Note.Music where
infixr 5 :+:, :=:
```

The previous chapters introduced some of the fundamental ideas of functional programming in Haskell. Also introduced were several of Euterpea's functions and operators, such as *note*, *rest*, (:+:), (:=:), and *trans*. This chapter will reveal the actual definitions of these functions and operators, thus exposing Euterpea's underlying structure and overall design at the note level. In addition, a number of other musical ideas will be developed, and in the process more Haskell features will be introduced as well.

2.1 Preliminaries

Sometimes it is convenient 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, as discussed in the last chapter, in music theory a pitch is defined as a pair, consisting of a *pitch class* and an *octave*. Assuming the existence of a data type called *PitchClass* (which we will return to shortly), we can write the following type synonym:

> **type** *Pitch* = (*PitchClass*, *Octave*)

For example, concert A (i.e. A440) corresponds to the pitch $(A, 4) :: \mathit{Pitch}$, and the lowest and highest notes on a piano correspond to $(A, 0) :: \mathit{Pitch}$ and $(C, 8) :: \mathit{Pitch}$, respectively.

Another important musical concept is *duration*. Rather than use either integers or floating-point numbers, Euterpea uses *rational* numbers to denote duration:

> **type** *Dur* = *Rational*

Rational is the data type of rational numbers expressed as ratios of *Integer*s in Haskell. The choice of *Rational* is somewhat subjective, but is justified by three observations: (1) many durations are expressed as ratios in music theory (5:4 rhythm, quarter notes, dotted notes, and so on), (2) *Rational* numbers are exact (unlike floating point numbers), which is important in many computer music applications, and (3) irrational durations are rarely needed.

Rational numbers in Haskell are printed by GHC 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 a program, however, once it is given the proper type, we can use the normal division operator, as in the following definition of 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 | Ff | Es | F | Gff | Ess | Fs
                | Gf | Fss | G | Aff | Gs | Af | Gss | A | Bff | As
                | Bf | Ass | B | Bs | Bss
```

Details: All of the names to the right of the equal sign in a **data** declaration are called *constructors*, and 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.

The *PitchClass* data type declaration essentially enumerates 35 pitch class names (five for each of the note names A through G). Note that both double-sharps and double-flats are included, resulting in many enharmonics (i.e., two notes that "sound the same," such as G♯ and A♭).

(The order of the pitch classes may seem a bit odd, but the idea is that if a pitch class pc_1 is to the left of a pitch class pc_2, then pc_1's pitch is "lower than" that of pc_2. This idea will be formalized and exploited in Chapter 7.1.)

Keep in mind that *PitchClass* is a completely new, user-defined data type that is not equal to any other. This is what distinguishes a **data** declaration from a **type** declaration. As another example of the use of a **data** declaration to define a simple enumerated type, Haskell's Boolean data type, called *Bool*, is predefined in Haskell simply as:

data *Bool* = *False* | *True*

2.2 Notes, Music, and Polymorphism

We can of course define other data types for other purposes. For example, we will want to define the notion of a *note* and a *rest*. Both of these can be thought of as "primitive" musical values, and thus as a first attempt we might write:

```
data Primitive = Note Dur Pitch
               | Rest Dur
```

Analogously to our previous data type declarations, the above declaration says that a *Primitive* is either a *Note* or a *Rest*. However, it is different in that the constructors *Note* and *Rest* take arguments, like functions do. In the case of *Note*, it takes two arguments, whose types are *Dur* and *Pitch*, respectively, whereas *Rest* takes one argument, a value of type *Dur*. In other words, the types of *Note* and *Rest* are:

```
Note :: Dur → Pitch → Primitive
Rest :: Dur →         Primitive
```

For example, *Note qn a440* 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 resolve this, Euterpea uses an important concept in Haskell, namely *polymorphism*—the ability to parameterize, or abstract, over types (*poly* means *many* and *morphism* refers to the structure, or *form*, of objects).

Primitive can be redefined as a *polymorphic data type* as follows. Instead of fixing the type of the pitch of a note, it is left unspecified through the use of a *type variable*:

```
data Primitive a = Note Dur a
                 | Rest Dur
```

Note that the type variable a is used as an argument to *Primitive*, and then used in the body of the declaration—just like a variable in a function. This version of *Primitive* is more general than the previous version—indeed, note that *Primitive Pitch* is the same as (or, technically, is *isomorphic to*) the previous version of *Primitive*. But additionally, *Primitive* is now more flexible than the previous version, since, for example, we can add loudness by pairing loudness with pitch, as in *Primitive* (*Pitch*, *Loudness*). Other concrete instances of this idea will be introduced later.

Details: Type variables such as a above must begin with a lower-case letter, to distinguish them from concrete types such as *Dur* or *Pitch*. Since *Primitive* takes an argument, it is called a *type constructor*, wherease *Note* and *Rest* are just called constructors (or value constructors).

Another way to interpret this data declaration is to say that for any type a, this declaration declares the types of its constructors to be:

```
Note :: Dur → a → Primitive a
Rest :: Dur →     Primitive a
```

Even though *Note* and *Rest* are called data constructors, they are still functions, and they have a type. Since they both have type variables in their type signatures, they are examples of *polymorphic functions*.

It is helpful to think of polymorphism as applying the abstraction principle at the type level—indeed it is often called *type abstraction*. Many more examples of both polymorphic functions and polymorphic data types will be explored in detail in Chapter 3.

So far Euterpea's primitive notes and rests have been introduced—but how do we combine many notes and rests into a larger composition? To achieve this, Euterpea defines another polymorphic data type, perhaps the most important data type used in this textbook, which defines the fundamental structure of a note-level musical entity:

```
data Music a =
    Prim (Primitive a)          -- primitive value
  | Music a :+: Music a         -- sequential composition
  | Music a :=: Music a         -- parallel composition
  | Modify Control (Music a)    -- modifier
```

Following the reasoning above, the types of these constructors are:

```
Prim   :: Primitive a           → Music a
(:+:)  :: Music a → Music a → Music a
(:=:)  :: Music a → Music a → Music a
Modify :: Control → Music a  → Music a
```

These four constructors then are also polymorphic functions.

Details: Note the use of the *infix constructors* (:+:) and (:=:). Infix constructors are just like infix operators in Haskell, but they must begin with a colon. This syntactic distinction makes it clear when pattern matching is intended, and is analogous to the distinction between ordinary names (which must begin with a lower-case character) and constructor names (which must begin with an upper-case character).

The observant reader will also recall that at the very beginning of this chapter—corresponding to the module containing all the code in this chapter—the following line appeared:

```
infixr 5 :+:, :=:
```

This is called a *fixity declaration*. The "r" after the word "**infix**" means that the specified operators—in this case (:+:) and (:=:)—are to have *right* associativity, and the "5" specifies their *precedence level* (these operators will bind more tightly than an operator with a lower precedence).

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

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

  $$a440m :: Music\ Pitch$$
  $$a440m = Prim\ (Note\ qn\ a440)$$

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

- m_1 :+: m_2 is the *sequential composition* of m_1 and m_2; i.e. m_1 and m_2 are played in sequence.

- m_1 :=: m_2 is the *parallel composition* of m_1 and m_2; i.e. m_1 and m_2 are played simultaneously. The duration of the result is the duration of the longer of m_1 and m_2.

  (Recall that these last two operators were introduced in the last chapter. You can see now that they are actually constructors of an algebraic data type.)

- *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 the data type is said to be *recursive* (analogous to a recursive function). 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.

It is convenient to represent these musical ideas as a recursive datatype because it allows us to not only *construct* musical values, but also take them apart, analyze their structure, print them in a structure-preserving way, transform them, interpret them for performance purposes, and so on. Many examples of these kinds of processes will be seen in this textbook.

The *Control* data type is used by the *Modify* constructor to annotate a *Music* value with a *tempo change*, a *transposition*, a *phrase attribute*, a *player name*, or an *instrument*. This data type is unimportant at the moment, but for completeness here is its full definition:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Control} = & & \\
\quad\ \ \mathit{Tempo} & \mathit{Rational} & \text{-- scale the tempo} \\
\quad \mid \mathit{Transpose} & \mathit{AbsPitch} & \text{-- transposition} \\
\quad \mid \mathit{Instrument} & \mathit{InstrumentName} & \text{-- instrument label} \\
\quad \mid \mathit{Phrase} & [\mathit{PhraseAttribute}] & \text{-- phrase attributes} \\
\quad \mid \mathit{Player} & \mathit{PlayerName} & \text{-- player label} \\
\quad \mid \mathit{KeySig} & \mathit{PitchClass}\ \mathit{Mode} & \text{-- key signature and mode}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{PlayerName} = \mathit{String} \\
\textbf{data}\ \mathit{Mode} \qquad\ \ = \mathit{Major} \mid \mathit{Minor}
\end{array}
$$

AbsPitch ("absolute pitch," to be defined in Section 2.4) is just a type synonym for *Int*. Instrument names are borrowed from the General MIDI standard [Ass13a, Ass13b], 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 8. The *KeySig* constructor attaches a key signature to a *Music* value, and is different conceptually from transposition.

2.3 Convenient Auxiliary Functions

For convenience, and in anticipation of their frequent use, a number of functions are defined in Euterpea to make it easier to write certain kinds of musical values. For starters:

$$
\begin{array}{ll}
\mathit{note} & :: \mathit{Dur} \to a \to \mathit{Music}\ a \\
\mathit{note}\ d\ p & = \mathit{Prim}\ (\mathit{Note}\ d\ p) \\
\mathit{rest} & :: \mathit{Dur} \to \mathit{Music}\ a \\
\mathit{rest}\ d & = \mathit{Prim}\ (\mathit{Rest}\ d) \\
\mathit{tempo} & :: \mathit{Dur} \to \mathit{Music}\ a \to \mathit{Music}\ a \\
\mathit{tempo}\ r\ m & = \mathit{Modify}\ (\mathit{Tempo}\ r)\ m \\
\mathit{transpose} & :: \mathit{AbsPitch} \to \mathit{Music}\ a \to \mathit{Music}\ a \\
\mathit{transpose}\ i\ m & = \mathit{Modify}\ (\mathit{Transpose}\ i)\ m \\
\mathit{instrument} & :: \mathit{InstrumentName} \to \mathit{Music}\ a \to \mathit{Music}\ a \\
\mathit{instrument}\ i\ m & = \mathit{Modify}\ (\mathit{Instrument}\ i)\ m \\
\mathit{phrase} & :: [\mathit{PhraseAttribute}] \to \mathit{Music}\ a \to \mathit{Music}\ a \\
\mathit{phrase}\ pa\ m & = \mathit{Modify}\ (\mathit{Phrase}\ pa)\ m \\
\mathit{player} & :: \mathit{PlayerName} \to \mathit{Music}\ a \to \mathit{Music}\ a
\end{array}
$$

```
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
```

Figure 2.1: General MIDI Instrument Names

$$
\begin{aligned}
&\mathit{player}\ \mathit{pn}\ m &&= \mathit{Modify}\ (\mathit{Player}\ \mathit{pn})\ m\\
&\mathit{keysig} &&:: \mathit{PitchClass} \to \mathit{Mode} \to \mathit{Music}\ a \to \mathit{Music}\ a\\
&\mathit{keysig}\ \mathit{pc}\ \mathit{mo}\ m &&= \mathit{Modify}\ (\mathit{KeySig}\ \mathit{pc}\ \mathit{mo})\ m
\end{aligned}
$$

Note that each of these functions is polymorphic, a trait inherited from the data types that it uses. Also recall that the first two of these functions were used in an example in the last chapter.

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.

Details: Figures 2.2 and 2.3 demonstrate that at the top level of a program, more than one equation can be placed on one line, as long as they are separated by a semicolon. This allows us to save vertical space on the page, and is useful whenever each line is relatively short. The semicolon is not needed at the end of a single equation, or at the end of the last equation on a line. This convenient feature is part of Haskell's *layout* rule, and will be explained in more detail later.

More than one equation can also be placed on one line in a **let** expression, as demonstrated below:

$$
\begin{aligned}
&\textbf{let}\ x = 1; y = 2\\
&\textbf{in}\ x + y
\end{aligned}
$$

2.3.1 A Simple Example

As a simple example, suppose we wish to generate a ii-V-I chord progression in a particular major key. In music theory, such a chord progression begins with a minor chord on the second degree of the major scale, followed by a major chord on the fifth degree, and ending in a major chord on the first degree. We can write this in Euterpea, using triads in the key of C major, as follows:

$$
\begin{aligned}
&\mathit{t251} :: \mathit{Music}\ \mathit{Pitch}\\
&\mathit{t251} = \textbf{let}\ \mathit{dMinor} = d\ 4\ \mathit{wn} :=: f\ 4\ \mathit{wn} :=: a\ 4\ \mathit{wn}\\
&\qquad\qquad\quad \mathit{gMajor} = g\ 4\ \mathit{wn} :=: b\ 4\ \mathit{wn} :=: d\ 5\ \mathit{wn}\\
&\qquad\qquad\quad \mathit{cMajor} = c\ 4\ \mathit{bn}\ :=: e\ 4\ \mathit{bn}\ :=: g\ 4\ \mathit{bn}\\
&\qquad\quad \textbf{in}\ \mathit{dMinor} :+: \mathit{gMajor} :+: \mathit{cMajor}
\end{aligned}
$$

```
cff, cf, c, cs, css, dff, df, d, ds, dss, eff, ef, e, es, ess, fff, ff, f,
  fs, fss, gff, gf, g, gs, gss, aff, af, a, as, ass, bff, bf, b, bs, bss ::
    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

```haskell
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, sfnr, 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    -- eighth 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 Duration and Rest Names

Details: Note that more than one equation is allowed in a **let** expression, just like at the top level of a program. 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 in 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 (such as a semicolon) to denote the end of one equation and the beginning of the next, thus enhancing readability. In practice, use of layout is rather intuitive. Just remember two things:

First, the first character following **let** (and a few other keywords that will be introduced 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, be sure that the starting column is further to the right than the starting column associated with any immediately surrounding **let** 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.

We can play this simple example using Euterpea's *play* function by simply typing:

play t251

at the GHCi command line. Default instruments and tempos are used to convert *t251* into MIDI and then play the result through your computer's standard sound card.

Details: It is important when using *play* that the type of its argument is made clear. In the case of *t251*, it is clear from the type signature in its definition. But for reasons to be explained in Chapter 7, if we write even something very simple such as $play\ (note\ qn\ (C, 4))$, Haskell cannot infer exactly what kind of number 4 is, and therefore cannot infer that $(C, 4)$ is intended to be a *Pitch*. We can get around this either by writing:

$$m :: Pitch$$
$$m = note\ qn\ (C, 4)$$

in which case $play\ m$ will work just fine, or we can include the type signature "in-line" with the expression, as in $play\ (note\ qn\ ((C, 4) :: Pitch))$.

Exercise 2.1 The above example is fairly concrete, in that, for one, it is rooted in C major, and furthermore it has a fixed tempo. Define a function $twoFiveOne :: Pitch \to Dur \to Music\ Pitch$ such that $twoFiveOne\ p\ d$ constructs a ii-V-I chord progression in the key whose major scale begins on the pitch p (i.e. the first degree of the major scale on which the progression is being constructed), where the duration of the first two chords is each d, and the duration of the last chord is $2 * d$.

To verify your code, prove by calculation that $twoFiveOne\ (C, 4)\ wn = t251$.

Exercise 2.2 The *PitchClass* data type implies the use of standard Western harmony, in particular the use of a *twelve-tone equal temperament scale*. But there are many other scale possibilities. For example, the *pentatonic blues scale* consists of five notes (thus "pentatonic") and, in the key of C, approximately corresponds to the notes C, E♭, F, G, and B♭. More abstractly, let's call these the root, minor third, fourth, fifth, and minor seventh, respectively. Your job is to:

1. Define a new algebraic data type called *BluesPitchClass* that captures this scale (for example, you may wish to use the constructor names *Ro*, *MT*, *Fo*, *Fi*, and *MS*).

2. Define a type synonym *BluesPitch*, akin to *Pitch*.

3. Define auxiliary functions *ro*, *mt*, *fo*, *fi*, and *ms*, akin to those in Figure 2.2, that make it easy to construct notes of type *Music BluesPitch*.

4. In order to play a value of type *Music BluesPitch* using MIDI, it will have to be converted into a *Music Pitch* value. Define a function *fromBlues* :: *Music BluesPitch* → *Music Pitch* to do this, using the "approximate" translation described at the beginning of this exercise.

   Hint: To do this properly, you will have to pattern match against the *Music* value, something like this:

   $$\begin{aligned}
   &\mathit{fromBlues}\ (\mathit{Prim}\ (\mathit{Note}\ d\ p)) &&= ... \\
   &\mathit{fromBlues}\ (\mathit{Prim}\ (\mathit{Rest}\ d)) &&= ... \\
   &\mathit{fromBlues}\ (m_1 \mathbin{:+:} m_2) &&= ... \\
   &\mathit{fromBlues}\ (m_1 \mathbin{:=:} m_2) &&= ... \\
   &\mathit{fromBlues}\ (\mathit{Modify}...) &&= ...
   \end{aligned}$$

5. Write out a few melodies of type *Music BluesPitch*, and play them using *fromBlues* and *play*.

2.4 Absolute Pitches

Treating pitches simply as integers is useful in many settings, so Euterpea uses a type synonym to define the concept of an "absolute pitch:"

$$\mathbf{type}\ \mathit{AbsPitch} = \mathit{Int}$$

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:

$$\begin{aligned}
&\mathit{absPitch} &&:: \mathit{Pitch} \rightarrow \mathit{AbsPitch} \\
&\mathit{absPitch}\ (\mathit{pc}, \mathit{oct}) &&= 12 * \mathit{oct} + \mathit{pcToInt}\ \mathit{pc}
\end{aligned}$$

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

pcToInt is a function that converts a particular pitch class to an index, easily but tediously expressed as shown in Figure 2.4. But there is a subtlety: according to music theory convention, pitches are assigned integers in the range 0 to 11, i.e. modulo 12, starting on pitch class C. In other words, the index of C is 0, C♭ is 11, and B♯ is 0. However, that would mean the absolute

$$
\begin{array}{l}
\mathit{pcToInt} :: \mathit{PitchClass} \to \mathit{Int} \\[4pt]
\begin{array}{lll}
\mathit{pcToInt}\ \mathit{Cff} = -2; & \mathit{pcToInt}\ \mathit{Dff} = 0; & \mathit{pcToInt}\ \mathit{Eff} = 2 \\
\mathit{pcToInt}\ \mathit{Cf} = -1; & \mathit{pcToInt}\ \mathit{Df} = 1; & \mathit{pcToInt}\ \mathit{Ef} = 3 \\
\mathit{pcToInt}\ C = 0; & \mathit{pcToInt}\ D = 2; & \mathit{pcToInt}\ E = 4 \\
\mathit{pcToInt}\ \mathit{Cs} = 1; & \mathit{pcToInt}\ \mathit{Ds} = 3; & \mathit{pcToInt}\ \mathit{Es} = 5 \\
\mathit{pcToInt}\ \mathit{Css} = 2; & \mathit{pcToInt}\ \mathit{Dss} = 4; & \mathit{pcToInt}\ \mathit{Ess} = 6 \\[4pt]
\mathit{pcToInt}\ \mathit{Fff} = 3; & \mathit{pcToInt}\ \mathit{Gff} = 5; & \mathit{pcToInt}\ \mathit{Aff} = 7 \\
\mathit{pcToInt}\ \mathit{Ff} = 4; & \mathit{pcToInt}\ \mathit{Gf} = 6; & \mathit{pcToInt}\ \mathit{Af} = 8 \\
\mathit{pcToInt}\ F = 5; & \mathit{pcToInt}\ G = 7; & \mathit{pcToInt}\ A = 9 \\
\mathit{pcToInt}\ \mathit{Fs} = 6; & \mathit{pcToInt}\ \mathit{Gs} = 8; & \mathit{pcToInt}\ \mathit{As} = 10 \\
\mathit{pcToInt}\ \mathit{Fss} = 7; & \mathit{pcToInt}\ \mathit{Gss} = 9; & \mathit{pcToInt}\ \mathit{Ass} = 11
\end{array} \\[4pt]
\mathit{pcToInt}\ \mathit{Bff} = 9 \\
\mathit{pcToInt}\ \mathit{Bf} = 10 \\
\mathit{pcToInt}\ B = 11 \\
\mathit{pcToInt}\ \mathit{Bs} = 12 \\
\mathit{pcToInt}\ \mathit{Bss} = 13
\end{array}
$$

Figure 2.4: Converting Pitch Classes to Integers

pitch of $(C, 4)$, say, would be 48, whereas $(\mathit{Cf}, 4)$ would be 59. Somehow the latter does not seem right—47 would be a more logical choice. Therefore the definition in Figure 2.4 is written in such a way that the wrap-round does not happen, i.e. numbers outside the range 0 to 11 are used. With this definition, $\mathit{absPitch}\ (\mathit{Cf}, 4)$ yields 47, as desired.

Details: The repetition of "*pcToInt*" above can be avoided by using a Haskell **case** expression, resulting in a more compact definition:

$$
\begin{array}{l}
\mathit{pcToInt} \quad :: \mathit{PitchClass} \rightarrow \mathit{Int} \\
\mathit{pcToInt}\ \mathit{pc} = \mathbf{case}\ \mathit{pc}\ \mathbf{of} \\
\quad \mathit{Cff} \rightarrow -2; \mathit{Cf} \rightarrow -1; \mathit{C} \rightarrow 0; \ \mathit{Cs} \rightarrow 1; \ \mathit{Css} \rightarrow 2; \\
\quad \mathit{Dff} \rightarrow 0; \ \mathit{Df} \rightarrow 1; \ \mathit{D} \rightarrow 2; \ \mathit{Ds} \rightarrow 3; \ \mathit{Dss} \rightarrow 4; \\
\quad \mathit{Eff} \rightarrow 2; \ \mathit{Ef} \rightarrow 3; \ \mathit{E} \rightarrow 4; \ \mathit{Es} \rightarrow 5; \ \mathit{Ess} \rightarrow 6; \\
\quad \mathit{Fff} \rightarrow 3; \ \mathit{Ff} \rightarrow 4; \ \mathit{F} \rightarrow 5; \ \mathit{Fs} \rightarrow 6; \ \mathit{Fss} \rightarrow 7; \\
\quad \mathit{Gff} \rightarrow 5; \ \mathit{Gf} \rightarrow 6; \ \mathit{G} \rightarrow 7; \ \mathit{Gs} \rightarrow 8; \ \mathit{Gss} \rightarrow 9; \\
\quad \mathit{Aff} \rightarrow 7; \ \mathit{Af} \rightarrow 8; \ \mathit{A} \rightarrow 9; \ \mathit{As} \rightarrow 10; \mathit{Ass} \rightarrow 11; \\
\quad \mathit{Bff} \rightarrow 9; \ \mathit{Bf} \rightarrow 10; \mathit{B} \rightarrow 11; \mathit{Bs} \rightarrow 12; \mathit{Bss} \rightarrow 13
\end{array}
$$

As you can see, a **case** expression allows multiple pattern-matches on an expression without using equations. Note that layout applies to the body of a case expression, and can be overriden as before using a semicolon. (As in a function type signature, the right-pointing arrow in a **case** expression must be typed as "->" on your computer keyboard.)

The body of a **case** expression observes layout just as a **let** expression, including the fact that semicolons can be used, as above, to place more than one pattern match on the same line.

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 $(Ef, 1)$. Euterpea takes the approach of always returning a sharp in such ambiguous cases:

$$
\begin{array}{l}
\mathit{pitch} \quad :: \mathit{AbsPitch} \rightarrow \mathit{Pitch} \\
\mathit{pitch}\ \mathit{ap} = \\
\quad \mathbf{let}\ (\mathit{oct}, n) = \mathit{divMod}\ \mathit{ap}\ 12 \\
\quad \mathbf{in}\ ([C, Cs, D, Ds, E, F, Fs, G, Gs, A, As, B]\ !!\ n, \mathit{oct})
\end{array}
$$

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.

Given *pitch* and *absPitch*, it is now easy to define a function *trans* that transposes pitches:

$$
\begin{aligned}
&trans \quad :: Int \rightarrow Pitch \rightarrow Pitch \\
&trans\ i\ p = pitch\ (absPitch\ p + i)
\end{aligned}
$$

With this definition, all of the operators and functions introduced in the previous chapter have been covered.

Exercise 2.3 Show that $abspitch\ (pitch\ ap) = ap$, and, up to enharmonic equivalences, $pitch\ (abspitch\ p) = p$.

Exercise 2.4 Show that $trans\ i\ (trans\ j\ p) = trans\ (i + j)\ p$.

Exercise 2.5 *Transpose* is part of the *Control* data type, which in turn is part of the *Music* data type. Its use in transposing a *Music* value is thus a kind of "annotation"—it doesn't really change the *Music* value, it just annotates it as something that is transposed.

Define instead a recursive function $transM :: AbsPitch \rightarrow Music\ Pitch \rightarrow Music\ Pitch$ that actually changes each note in a *Music Pitch* value by transposing it by the interval represented by the first argument.

Hint: To do this properly, you will have to pattern match against the *Music* value, something like this:

$$
\begin{aligned}
&transM\ ap\ (Prim\ (Note\ d\ p)) &&= ... \\
&transM\ ap\ (Prim\ (Rest\ d)) &&= ... \\
&transM\ ap\ (m_1 :+: m_2) &&= ... \\
&transM\ ap\ (m_1 :=: m_2) &&= ... \\
&transM\ ap\ (Modify ...) &&= ...
\end{aligned}
$$

Chapter 3

Polymorphic and Higher-Order Functions

Several examples of polymorphic data types were introduced in the last couple of chapters. In this chapter the focus is on *polymorphic functions*, which are most commonly defined over polymorphic data types.

The already familiar *list* is the protoypical example of a polymorphic data type, and it will be studied 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. But in addition the *Music* data type is polymorphic, and several new functions that operate on it polymorphiccally will also be defined,

(A more detailed discussion of predefined polymorphic functions that operate on lists can be found in Appendix A.)

This chapter also introduces *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). Higher-order functions permit the elegant and concise expression of many musical concepts. Together with polymorphism, higher-order functions substantially increase the programmer's expressive power and ability to reuse code.

Both of these new ideas follow naturally the foundations that have already been established.

3.1 Polymorphic Types

In previous chapters, examples of lists containing several different kinds of elements—integers, characters, pitch classes, and so on—were introduced, and we can well imagine situations requiring lists of other element types. Sometimes, however, it is not necessary to be so particular about the type of the elements. For example, suppose we wish to define a function *length* that determines the number of elements in a list. It does not really matter whether the list contains integers, pitch classes, or even other lists—we can imagine computing the length in exactly the same way in each case. The obvious definition is:

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

This recursive definition is self-explanatory. Indeed, 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, we would like to say that, for *any* type a, the type of *length* is $[\,a\,] \rightarrow \mathit{Integer}$. In mathematics we might write this as:

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

But in Haskell this is written simply as:

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

In other words, the universal quantification of the type variable a is implicit.

> **Details:** Generic names for types, such as a above, are called *type variables*, and are uncapitalized to distinguish them from concrete types such as *Integer*.

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

$$\begin{aligned} &\mathit{length}\ [1,2,3] &&\Longrightarrow 3 \\ &\mathit{length}\ [\,C, D, \mathit{Ef}\,] \Longrightarrow 3 \\ &\mathit{length}\ [[1],[\,],[2,3,4]] \Longrightarrow 3 \end{aligned}$$

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 predefined in Haskell:

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

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. Note that we could (perhaps should) have used a wildcard in place of the variable x in the definition of *length*.

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

$$\begin{aligned}
&\mathit{head}\ [1, 2, 3] \Rightarrow 1 \\
&\mathit{head}\ [C, D, \mathit{Ef}\,] \Rightarrow C \\
&\mathit{tail}\ [1, 2, 3] \quad \Rightarrow [2, 3] \\
&\mathit{tail}\ [C, D, \mathit{Ef}\,] \quad \Rightarrow [D, \mathit{Ef}\,]
\end{aligned}$$

Note that, for any non-empty list *xs*, *head* and *tail* obey the following law:

$$\mathit{head}\ \mathit{xs} : \mathit{tail}\ \mathit{xs} = \mathit{xs}$$

Functions such as *length*, *head*, and *tail* are said to be *polymorphic*. 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

Given a list of pitches, suppose we wish to convert each pitch into an absolute pitch. We could define a function:

$$\begin{aligned}
&\mathit{toAbsPitches} && :: [\mathit{Pitch}] \rightarrow [\mathit{AbsPitch}] \\
&\mathit{toAbsPitches}\ [\,] && = [\,] \\
&\mathit{toAbsPitches}\ (p : \mathit{ps}) && = \mathit{absPitch}\ p : \mathit{toAbsPitches}\ \mathit{ps}
\end{aligned}$$

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

$$
\begin{aligned}
&\mathit{toPitches} && :: [\mathit{AbsPitch}] \rightarrow [\mathit{Pitch}] \\
&\mathit{toPitches}\ [\,] && = [\,] \\
&\mathit{toPitches}\ (a : \mathit{as}) && = \mathit{pitch}\ a : \mathit{toPitches}\ \mathit{as}
\end{aligned}
$$

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 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, recall that it is sometimes helpful to first identify those things that *are not* 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*; consider them instances of a new name, f. Rewriting either of the above functions as a new function—call it *map*—that takes an extra argument f, yields:

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

This recursive pattern of operations is so common that *map* is predefined in Haskell (and is why the name *map* was chosen in the first place).

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

$$
\begin{aligned}
&\mathit{toAbsPitches} && :: [\mathit{Pitch}] \rightarrow [\mathit{AbsPitch}] \\
&\mathit{toAbsPitches}\ \mathit{ps} && = \mathit{map}\ \mathit{absPitch}\ \mathit{ps}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{toPitches} && :: [\mathit{AbsPitch}] \rightarrow [\mathit{Pitch}] \\
&\mathit{toPitches}\ \mathit{as} && = \mathit{map}\ \mathit{pitch}\ \mathit{as}
\end{aligned}
$$

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. Inductive proofs are discussed in detail, including for these two examples, in Chapter 10.

3.2.1 Map is Polymorphic

What should the type of *map* be? Looking first at its use in *toAbsPitches*, note that it takes the function *absPitch* :: *Pitch* → *AbsPitch* as its first argument and a list of *Pitch*s as its second argument, returning a list of *AbsPitch*s as its result. So its type must be:

$$map :: (Pitch \to AbsPitch) \to [\,Pitch\,] \to [\,AbsPitch\,]$$

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

$$map :: (AbsPitch \to Pitch) \to [\,AbsPitch\,] \to [\,Pitch\,]$$

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:

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

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.

Note, as we did in Section 2.2, that the above reasoning can be viewed as the abstraction principle at work at the type level.

Details: In Chapter 1 it was mentioned that every expression in Haskell has an associated type. But with polymorphism, we might wonder if there is just one type for every expression. For example, *map* could have any of these types:

$$
\begin{array}{l}
(a \to b) \to [a] \to [b] \\
(\mathit{Integer} \to b) \to [\mathit{Integer}] \to [b] \\
(a \to \mathit{Float}) \to [a] \to [\mathit{Float}] \\
(\mathit{Char} \to \mathit{Char}) \to [\mathit{Char}] \to [\mathit{Char}]
\end{array}
$$

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 $a \to b \to c$. As another example, the principal type of *head* is $[a] \to a$; the types $[b] \to a$, $b \to a$, or even a are too general, whereas something like $[\mathit{Integer}] \to \mathit{Integer}$ is too specific. (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 several other functional languages [Hud89].)

3.2.2 Using map

For a musical example involving *map*, consider the task of generating a six-note whole-tone scale starting at a given pitch:[1]

$$
\begin{array}{l}
\mathit{wts} \quad :: \mathit{Pitch} \to [\mathit{Music}\ \mathit{Pitch}] \\
\mathit{wts}\ p = \textbf{let}\ f\ \mathit{ap} = \mathit{note}\ \mathit{qn}\ (\mathit{pitch}\ (\mathit{absPitch}\ p + \mathit{ap})) \\
\qquad\qquad \textbf{in}\ \mathit{map}\ f\ [0,2,4,6,8]
\end{array}
$$

For example:

$$
\begin{array}{l}
\mathit{wts}\ \mathit{a440} \\
\Longrightarrow [\mathit{note}\ \mathit{qn}\ (A,4), \quad \mathit{note}\ \mathit{qn}\ (B,4), \mathit{note}\ \mathit{qn}\ (C\#,4), \\
\qquad \mathit{note}\ \mathit{qn}\ (D\#,4), \mathit{note}\ \mathit{qn}\ (F,4), \mathit{note}\ \mathit{qn}\ (G,4)]
\end{array}
$$

Exercise 3.1 Using *map*, define:

1. A function $f_1 :: \mathit{Int} \to [\mathit{Pitch}] \to [\mathit{Pitch}]$ that transposes each pitch in its second argument by the amount specified in its first argument.

[1] A whole-tone scale is a sequence of six ascending notes, with each adjacent pair of notes separated by two semitones, i.e. a whole note.

2. A function $f_2 :: [\,Dur\,] \rightarrow [\,Music\ a\,]$ that turns a list of durations into a list of rests, each having the corresponding duration.

3. A function $f_3 :: [\,Music\ Pitch\,] \rightarrow [\,Music\ Pitch\,]$ that takes a list of music values (that are assumed to be single notes), and for each such note, halves its duration and places a rest of that same duration after it. For example:

$$\begin{array}{l} f_3\ [\,c\ 4\ qn, d\ 4\ en, e\ 4\ hn\,] \\ \Longrightarrow [\,c\ 4\ en :+: rest\ en, d\ 4\ sn :+: rest\ sn, e\ 4\ qn :+: rest\ qn\,] \end{array}$$

   You can think of this as giving a staccato interpretation of the notes.

3.3 Append

Consider now 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 can define this as a polymorphic infix operator $(+\!\!+)$:

$$(+\!\!+) :: [\,a\,] \rightarrow [\,a\,] \rightarrow [\,a\,]$$

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

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

As usual, we can approach this problem by considering the various possibilities that could arise as input. But in the case of $(+\!\!+)$ there are *two* inputs—so which should be considered first? In general this is not an easy question to answer, so we could just try 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 $(+\!\!+)$. The full definition is thus:

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

Details: Note that an infix operator can be defined just as any other function, including pattern-matching, except that on the left-hand-side it is written using its infix syntax.

Also be aware that this textbook takes liberty in typesetting by displaying the append operator as $+\!\!+$. When you type your code, however, you will need to write `++`. Recall that infix operators in Haskell must not contain any numbers or letters of the alphabet, and also must not begin with a colon (because those are reserved to be infix constructors).

If we were to consider instead the second list first, then the first equation would still be easy:

$$xs +\!\!+ [\,] = xs$$

but the second is not so obvious:

$$xs +\!\!+ (y : ys) = ??$$

So it seems that the right choice was made to begin with.

Like *map*, the concatenation operator $(+\!\!+)$ is used so often that it is predefined in Haskell.

3.3.1 [Advanced] The Efficiency and Fixity of Append

In Chapter 10 the following simple property about $(+\!\!+)$ will be proved:

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

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:

$$2 * length\ xs + length\ ys$$

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

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 a more dramatic example of this property will be introduced.

3.4 Fold

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

$$\begin{aligned}
&line && :: [Music\ a] \to Music\ a \\
&line\ [\,] && = rest\ 0 \\
&line\ (m : ms) && = m \mathbin{:+:} line\ ms
\end{aligned}$$

Note that this function is polymorphic—the first example so far, in fact, of a polymorphic function involving the *Music* data type.

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

$$\begin{aligned}
&chord && :: [Music\ a] \to Music\ a \\
&chord\ [\,] && = rest\ 0 \\
&chord\ (m : ms) && = m \mathbin{:=:} chord\ ms
\end{aligned}$$

This function is also polymorphic.

In a completely different context we might wish to compute the highest pitch in a list of pitches, which we might capture in the following way:

$$\begin{aligned}
&maxPitch && :: [Pitch] \to Pitch \\
&maxPitch\ [\,] && = pitch\ 0 \\
&maxPitch\ (p : ps) && = p\ !!!\ maxPitch\ ps
\end{aligned}$$

where (!!!) is defined as:

$$p_1\ !!!\ p_2 = \textbf{if}\ absPitch\ p_1 > absPitch\ p_2\ \textbf{then}\ p_1\ \textbf{else}\ p_2$$

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 used earlier to discover *map*, we first identify those things that are changing. There are two situations: the *rest* 0 and *pitch* 0 values (for which the generic name *init*, for "initial value," will be used), and the (:+:), (:=:), and (!!!) operators (for which the generic name *op*, for "operator," will be used). Now rewriting any of the above three functions as a new function—call it *fold*—that takes extra arguments *op* and *init*, we arrive at:[2]

$$\begin{aligned}
&fold\ op\ init\ [\,] &&= init \\
&fold\ op\ init\ (x : xs) &&= x\ `op`\ fold\ op\ init\ xs
\end{aligned}$$

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 is being abstracted, but could also have been written $op\ x\ (fold\ op\ init\ xs)$.

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

$$\begin{aligned}
&line, chord :: [Music\ a] \rightarrow Music\ a \\
&line\ \ \ \ ms = fold\ (:+:)\ (rest\ 0)\ ms \\
&chord\ ms = fold\ (:=:)\ (rest\ 0)\ ms
\end{aligned}$$

$$\begin{aligned}
&maxPitch &&:: [Pitch] \rightarrow Pitch \\
&maxPitch\ ps &&= fold\ (!!!)\ (pitch\ 0)\ ps
\end{aligned}$$

[2] The use of the name "*fold*" for this function is historical (within the functional programming community), and has nothing to do with the use of "fold" and "unfold" in Chapter 1 to describe steps in a calculation.

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 apply (!!!) to $fold$ and $0\ ps$, which is nonsensical and ill-typed.)

In Chapter 10 we will use induction to prove that these new definitions are equivalent to the old.

fold, like *map*, is a highly useful—reusable—function, as will be seen 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:

$$fold :: (a \to b \to b) \to b \to [a] \to b$$

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

As a final example, recall the definition of *hList* from Chapter 1:

$$\begin{aligned}
&hList && :: Dur \to [Pitch] \to Music\ Pitch \\
&hList\ d\ [\,] && = rest\ 0 \\
&hList\ d\ (p : ps) && = hNote\ d\ p :\!+\!: hList\ d\ ps
\end{aligned}$$

A little thought should convince the reader that this can be rewritten as:

$$\begin{aligned}
hList\ d\ ps = &\ \mathbf{let}\ f\ p = hNote\ d\ p \\
&\ \mathbf{in}\ line\ (map\ f\ ps)
\end{aligned}$$

This version is more modular, in that it avoids explicit recursion, and is instead built up from smaller building blocks, namely *line* (which uses *fold*) and *map*.

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") whose definition is the same as that of *fold* given earlier:

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

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

$$\begin{array}{l} \mathit{foldr}\ \mathit{op}\ \mathit{init}\ (x_1 : x_2 : ... : xn : [\,]) \\ \Longrightarrow x_1\ \text{`}\mathit{op}\text{`}\ (x_2\ \text{`}\mathit{op}\text{`}\ (...(xn\ \text{`}\mathit{op}\text{`}\ \mathit{init})...)) \end{array}$$

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

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

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

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

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

$$\begin{array}{l} \mathit{foldl}\ \mathit{op}\ \mathit{init}\ (x_1 : x_2 : ... : xn : [\,]) \\ \Longrightarrow (...((\mathit{init}\ \text{`}\mathit{op}\text{`}\ x_1)\ \text{`}\mathit{op}\text{`}\ x_2)...)\ \text{`}\mathit{op}\text{`}\ xn \end{array}$$

3.4.2 [Advanced] 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 have two versions of *fold*? It turns out that there are situations where using one is more efficient, and possibly "more defined" (that is, the function terminates on more values of its input domain) than the other.

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

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

For example:

[3]This will be formally proved in Chapter 10.

$$\begin{aligned}&\mathit{concat}\ [[1],[3,4],[\,],[5,6]]\\ &\Longrightarrow [1] \mathbin{+\!\!+} ([3,4] \mathbin{+\!\!+} ([\,] \mathbin{+\!\!+} ([5,6] \mathbin{+\!\!+} [\,])))\\ &\Longrightarrow [1,3,4,5,6]\end{aligned}$$

More generally, we have that:

$$\begin{aligned}&\mathit{concat}\ [xs_1, xs_2, ..., xsn]\\ &\Rightarrow\ \ \mathit{foldr}\ (\mathbin{+\!\!+})\ [\,]\ [xs_1, xs_2, ..., xsn]\\ &\Longrightarrow xs_1 \mathbin{+\!\!+} (xs_2 \mathbin{+\!\!+} (...(xn \mathbin{+\!\!+} [\,]))...)\end{aligned}$$

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*, and there are n lists, then this cost is $(n-1) * \mathit{len}$.

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

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

then:

$$\begin{aligned}&\mathit{slowConcat}\ [xs_1, xs_2, ..., xsn]\\ &\Rightarrow\ \ \mathit{foldl}\ (\mathbin{+\!\!+})\ [\,]\ [xs_1, xs_2, ..., xsn]\\ &\Longrightarrow (...(([\,] \mathbin{+\!\!+} x_1) \mathbin{+\!\!+} x_2)...) \mathbin{+\!\!+} xn\end{aligned}$$

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

$$\begin{aligned}&\mathit{len} + (\mathit{len} + \mathit{len}) + (\mathit{len} + \mathit{len} + \mathit{len}) + ... + (n-1) * \mathit{len}\\ &= n * (n-1) * \mathit{len}/2\end{aligned}$$

which is considerably worse than $(n-1) * \mathit{len}$. 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

In certain contexts it may be understood that the functions *line* and *chord* should not be applied to an empty list. For such situations the Standard Prelude provides functions *foldr1* and *foldl1*, which return an error if applied to an empty list. And thus we may also desire to define versions of *line* and *chord* that adopt this behavior:

$$\begin{aligned}
&\mathit{line}_1, \mathit{chord1} :: [\mathit{Music}\ a] \to \mathit{Music}\ a\\
&\mathit{line}_1\ \mathit{ms} \quad\ = \mathit{foldr1}\ (:+:)\ \mathit{ms}\\
&\mathit{chord1}\ \mathit{ms} \quad = \mathit{foldr1}\ (:=:)\ \mathit{ms}
\end{aligned}$$

Note that *foldr1* and *foldl1* do not take an *init* argument.

In the case of *maxPitch* we could go a step further and say that the previous definition is in fact flawed, for who is to say what the maximum pitch of an empty list is? The choice of 0 was indeed arbitrary, and in a way it is nonsensical—how can 0 be the maximum if it is not even in the list? In such situations we might wish to define only one function, and to have that function return an error when presented with an empty list. For consistency with *line* and *chord*, however, that function is defined here with a new name:

$$\begin{aligned}
&\mathit{maxPitch1} \quad\ \ :: [\mathit{Pitch}] \to \mathit{Pitch}\\
&\mathit{maxPitch1}\ \mathit{ps} = \mathit{foldr1}\ (!!!)\ \mathit{ps}
\end{aligned}$$

3.5 [Advanced] A Final Example: Reverse

As a final example of a useful list function, consider the problem of *reversing* a list, which will be captured in a function called *reverse*. This could be useful, for example, when constructing the *retrograde* of a musical passage, i.e. the music as if it were played backwards. For example, $\mathit{reverse}\ [C, D, \mathit{Ef}]$ is $[\mathit{Ef}, D, C]$.

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 thus:

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

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 recall that the number of steps needed to compute $\mathit{xs} +\!\!+ \mathit{ys}$ 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? In fact, yes.

There is another algorithm for reversing a list, which can be described intuitively as follows: 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 the end of the original list is reached. 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 the need for a function that takes *two* arguments—the original list and the auxiliary one—yet *reverse* only takes one. This can be solved by creating an auxiliary function *rev*:

$$\begin{aligned}
&\mathit{reverse}\ \mathit{xs} = \mathbf{let}\ \mathit{rev}\ \mathit{acc}\ [\,] \qquad\quad = \mathit{acc}\\
&\qquad\qquad\qquad\quad\ \mathit{rev}\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{rev}\ (x : \mathit{acc})\ \mathit{xs}\\
&\qquad\qquad\quad\ \mathbf{in}\ \mathit{rev}\ [\,]\ \mathit{xs}
\end{aligned}$$

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*:

$$\begin{aligned}
&\mathit{foldl}\ \mathit{op}\ \mathit{init}\ [\,] \qquad\quad = \mathit{init}\\
&\mathit{foldl}\ \mathit{op}\ \mathit{init}\ (x : \mathit{xs}) = \mathit{foldl}\ \mathit{op}\ (\mathit{init}\ \text{`}\mathit{op}\text{`}\ x)\ \mathit{xs}
\end{aligned}$$

They are somewhat similar. In fact, suppose we were to slightly revise the definition of *rev* as follows:

$$\begin{aligned}
&\mathit{rev}\ \mathit{op}\ \mathit{acc}\ [\,] \qquad\quad = \mathit{acc}\\
&\mathit{rev}\ \mathit{op}\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{rev}\ \mathit{op}\ (\mathit{acc}\ \text{`}\mathit{op}\text{`}\ x)\ \mathit{xs}
\end{aligned}$$

Now *rev* looks strongly 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:

$$\mathit{revOp}\ a\ b = b : a$$

For note that:

$$\begin{aligned}
&\mathit{acc}\ \text{`}\mathit{revOp}\text{`}\ x\\
&\Rightarrow \mathit{revOp}\ \mathit{acc}\ x\\
&\Rightarrow x : \mathit{acc}
\end{aligned}$$

So *reverse* can be rewritten as:

$$\begin{array}{l} \mathit{reverse}\ \mathit{xs} = \textbf{let}\ \mathit{rev}\ \mathit{op}\ \mathit{acc}\ [\,] \qquad\quad = \mathit{acc} \\ \qquad\qquad\qquad\quad\ \mathit{rev}\ \mathit{op}\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{rev}\ \mathit{op}\ (\mathit{acc}\ \text{`}\mathit{op}\text{`}\ x)\ \mathit{xs} \\ \qquad\qquad\quad\ \ \textbf{in}\ \mathit{rev}\ \mathit{revOp}\ [\,]\ \mathit{xs} \end{array}$$

which is the same as:

$$\mathit{reverse}\ \mathit{xs} = \mathit{foldl}\ \mathit{revOp}\ [\,]\ \mathit{xs}$$

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

3.6 Currying

We can improve further upon some of the definitions given in this chapter using a technique called *currying simplification*. To understand this idea, first look closer at the notation used to write function applications, such as $\mathit{simple}\ x\ y\ z$. Although, as noted earlier, this is similar to the mathematical notation $\mathit{simple}(x, y, z)$, in fact there is an important difference, namely that $\mathit{simple}\ x\ y\ z$ is actually shorthand for $(((\mathit{simple}\ x)\ y)\ z)$. In other words, function application is *left associative*, taking one argument at a time.

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

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

$$\mathit{multSumByFive} = \mathit{simple}\ 5$$

What is *simple* 5? From the above argument it is clear that it must be a function. And from the definition of *simple* in Section 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.[4] It is helpful to look at the types of the intermediate functions as arguments are applied:

$$\begin{array}{ll} \mathit{simple} & :: \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Float} \\ \mathit{simple}\ 5 & :: \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Float} \\ \mathit{simple}\ 5\ a & :: \mathit{Float} \rightarrow \mathit{Float} \\ \mathit{simple}\ 5\ a\ b & :: \mathit{Float} \end{array}$$

For a musical example of this idea, recall the function $\mathit{note} :: \mathit{Dur} \rightarrow \mathit{Pitch} \rightarrow \mathit{Music}\ \mathit{Pitch}$. So $\mathit{note}\ \mathit{qn}$ is a function that, given a pitch, yields a quarter note rendition of that pitch. A common use of this idea is simplifying something like:

$$\mathit{note}\ \mathit{qn}\ p_1 :+: \mathit{note}\ \mathit{qn}\ p_2 :+: ... :+: \mathit{note}\ \mathit{qn}\ \mathit{pn}$$

to:

$$\mathit{line}\ (\mathit{map}\ (\mathit{note}\ \mathit{qn})\ [p_1, p_2, ..., \mathit{pn}])$$

Indeed, this idea is used extentively in the larger example in the next chapter.

3.6.1 Currying Simplification

We can also use currying to improve some of the previous function definitions as follows. Suppose that the values of $f\ x$ and $g\ x$ are the same, for all values of x. Then it seems clear that the functions f and g are equivalent.[5] So, if we wish to define f in terms of g, instead of writing:

$$f\ x = g\ x$$

We could instead simply write:

$$f = g$$

We can apply this reasoning to the definitions of *line* and *chord* from Section 3.4:

$$\begin{array}{ll} \mathit{line}\ \mathit{ms} & = \mathit{fold}\ (:+:)\ (\mathit{rest}\ 0)\ \mathit{ms} \\ \mathit{chord}\ \mathit{ms} & = \mathit{fold}\ (:=:)\ (\mathit{rest}\ 0)\ \mathit{ms} \end{array}$$

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

$$\begin{array}{ll} \mathit{line}\ \mathit{ms} & = (\mathit{fold}\ (:+:)\ (\mathit{rest}\ 0))\ \mathit{ms} \\ \mathit{chord}\ \mathit{ms} & = (\mathit{fold}\ (:=:)\ (\mathit{rest}\ 0))\ \mathit{ms} \end{array}$$

But now applying the same reasoning here as was used for f and g above

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

[5] In mathematics, we would say that the two functions are *extensionally equivalent*.

means that we can write these simply as:

$$\begin{aligned} &line &&= fold\ (:\!+\!:)\ (rest\ 0) \\ &chord &&= fold\ (:\!=\!:)\ (rest\ 0) \end{aligned}$$

Similarly, the definitions of *toAbsPitches* and *toPitches* from Section 3.2:

$$\begin{aligned} &toAbsPitches\ ps &&= map\ absPitch\ ps \\ &toPitches\ \ \ \ \ \ as &&= map\ pitch\ as \end{aligned}$$

can be rewritten as:

$$\begin{aligned} &toAbsPitches &&= map\ absPitch \\ &toPitches &&= map\ pitch \end{aligned}$$

Furthermore, the definition *hList*, most recently defined as:

$$\begin{aligned} hList\ d\ ps = &\ \mathbf{let}\ f\ p = hNote\ d\ p \\ &\ \mathbf{in}\ line\ (map\ f\ ps) \end{aligned}$$

can be rewritten as:

$$\begin{aligned} hList\ d\ ps = &\ \mathbf{let}\ f = hNote\ d \\ &\ \mathbf{in}\ line\ (map\ f\ ps) \end{aligned}$$

and since the definition of f is now so simple, we might as well “in-line” it:

$$hList\ d\ ps = line\ (map\ (hNote\ d)\ ps)$$

This kind of simplification will be referred to as “currying simplification” or just “currying.”[6]

> **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 remaining x on the right-hand side would become undefined!

3.6.2 [Advanced] Simplification of *reverse*

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*:

$$\begin{aligned} reverse\ xs = &\ \mathbf{let}\ revOp\ acc\ x = x : acc \\ &\ \mathbf{in}\ foldl\ revOp\ [\,]\ xs \end{aligned}$$

[6] In the Lambda Calculus this is called “eta contraction.”

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

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

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

Exercise 3.3 What is the type of *ys* in:

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

Exercise 3.4 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 Chapter 1.

Exercise 3.5 Define a function *applyAll* that, given a list of functions $[f_1, f_2, ..., fn]$ and a value v, returns the result $f_1\ (f_2\ (...(fn\ v)...))$. For example:

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

Exercise 3.6 Recall the discussion about the efficiency of $(+\!\!+)$ and *concat* in Chapter 3. Which of the following functions is more efficient, and why?

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

3.7 Errors

The last section suggested 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 all we want to do is 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 $error :: String \rightarrow a$. 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 that was discussed in Section 1.4. 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.

Exercise 3.7 Rewrite the definition of *length* non-recursively.

Exercise 3.8 Define a function that behaves as each of the following:

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

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

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

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

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

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

d) Adds "pointwise" the elements of a list of pairs. For example:

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

Exercise 3.9 Define a polymorphic function $\mathit{fuse} :: [\mathit{Dur}] \rightarrow [\mathit{Dur} \rightarrow \mathit{Music}\ a] \rightarrow [\mathit{Music}\ a]$ that combines a list of durations with a list of notes lacking a duration, to create a list of complete notes. For example:

$$\begin{array}{l} \mathit{fuse}\ [qn, hn, sn]\ [c\ 4, d\ 4, e\ 4] \\ \Longrightarrow [c\ 4\ qn, d\ 4\ hn, e\ 4\ sn] \end{array}$$

You may signal an error if the lists have unequal lengths.

In the next two exercises, give both recursive and (if possible) non-recursive definitions, and be sure to include type signatures.

Exercise 3.10 Define a function *maxAbsPitch* that determines the maximum absolute pitch of a list of absolute pitches. Define *minAbsPitch* analogously. Both functions should return an error if applied to the empty list.

Exercise 3.11 Define a function $\mathit{chrom} :: \mathit{Pitch} \rightarrow \mathit{Pitch} \rightarrow \mathit{Music}\ \mathit{Pitch}$ such that $\mathit{chrom}\ p_1\ p_2$ is a chromatic scale of quarter-notes whose first pitch is p_1 and last pitch is p_2. If $p_1 > p_2$, the scale should be descending, otherwise it should be ascending. If $p_1 == p_2$, 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 3.12 Abstractly, a scale can be described by the intervals between successive notes. For example, the 7-note major scale can be defined as the sequence of 6 intervals $[2, 2, 1, 2, 2, 2]$, and the 12-note chromatic scale by the 11 intervals $[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$. Define a function $\mathit{mkScale} :: \mathit{Pitch} \rightarrow [\mathit{Int}] \rightarrow \mathit{Music}\ \mathit{Pitch}$ such that $\mathit{mkScale}\ p\ \mathit{ints}$ is the scale beginning at pitch p and having the intervallic structure *ints*.

Exercise 3.13 Define an enumerated data type that captures each of the standard major scale modes: Ionian, Dorian, Phrygian, Lydian, Mixolydian, Aeolian, and Locrian. Then define a function *genScale* that, given one of these contructors, generates a scale in the intervalic form described in Exercise 3.12.

Exercise 3.14 Write the melody of “Frère Jacques” (or, “Are You Sleeping”) in Euterpea. Try to make it as succinct as possible. Then, using functions already defined, generate a traditional four-part round, i.e. four identical voices, each delayed successively by two measures. Use a different instrument to realize each voice.

Exercise 3.15 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.

Details: 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. However, the standard keyboard is adequately represented by a standard byte (eight bits), and thus we only need to consider the first 256 codes in the unicode representation. For the above 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.

Chapter 4

A Musical Interlude

At this point enough detail about Haskell and Euterpea has been covered that it is worth developing a small but full application or two. In this chapter an existing composition will be transcribed into Euterpea, thus exemplifying how to express conventional musical ideas in Euterpea. Then a simple form of algorithmic composition will be presented, where it will become apparent that more exotic things can be easily expressed as well.

But before tackling either of these, Haskell's *modules* will be described in more detail.

4.1 Modules

Haskell programs are partitioned into *modules* that capture common types, functions, etc. that naturally comprise an application. The first part of a module is called the module *header*, which declares what the name of the module is, and what other modules it might import. For this chapter the module's name is *Interlude*, into which the module *Euterpea* is imported:

```
module Interlude where
import Euterpea
```

> **Details:** Module names must always be capitalized (just like type names).

Maintaining the name space of modules in a large software system can be a daunting task. So Haskell provides a way to structure module names *hierachically*. Indeed, because the *Interlude* module is part of the overall Euterpea library, the actual module declaration that is used is:

```
module Euterpea.Examples.Interlude where
import Euterpea
```

This says that the *Interlude* module is part of the *Examples* folder in the overall *Euterpea* library. In general, these hierarchical names correspond to the folder (directory) structure of a particular implementation. Similarly, the name of the file containing the module is generally the same as the module name, plus the file extension (in this case, the name of the file is *Interlude.lhs*).

If we wish to use this module in another module M, say, it may be imported into M, just as was done above in importing *Euterpea* into *Interlude*:

```
module M where
import Euterpea.Examples.Interlude
```

This will make available in M all of the names of functions, types, and so on that are defined at the top-level of *Interlude*.

But this is not always what the programmer would like. Another purpose of a module is to manage the overall name space of an application. Modules allow us to structure an application in such a way that only the functionality intended for the end user is visible—everything else needed to implement the system is effectively hidden. In the case of *Interlude*, there are only two names whose visibillity is desirable: *childSong6*, and *prefix*. This can be achieved by writing the module header as follows:

```
module Euterpea.Examples.Interlude (childSong6, prefix) where
import Euterpea
```

This set of visible names is sometimes called the *export list* of the module. If the list is omitted, as was done initially, then *all* names defined at the top level of the module are exported.

Although explicit type signatures in export lists are not allowed, it is sometime useful to add them as comments, at least, as in:

```
module Euterpea.Examples.Interlude
          ( childSong6,   -- :: Music Pitch,
            prefix        -- :: [Music a] -¿ Music a)
          ) where
import Euterpea
```

In this case the list of names is sometimes called the *interface* to the module.

There are several other rules concerning the import and export of names to and from modules. Rather than introduce them all at once, they will be introduced as needed in future chapters.

4.2 Transcribing an Existing Score

Figure 4.1 shows the first 28 bars of Chick Corea's *Children's Songs No. 6*, written for electric piano [Cor94]. Analyzing the structure of this tune explores several basic issues that arise in the transcription of an existing score into Euterpea, including repeating phrases, grace notes, triplets, tempo, and specifying an instrument. To begin, however, we will define a couple of auxiliary functions to make our job easier.

4.2.1 Auxiliary Functions

For starters, note that there are several repeating patterns of notes in this composition, each enclosed in a rectangle in Figure 4.1. In fact, the bass line consists *entirely* of three repeating phrases. In anticipation of this, a function can be defined that repeats a phrase a particular number of times:

$$
\begin{aligned}
&\mathit{timesM} && :: \mathit{Int} \to \mathit{Music}\ a \to \mathit{Music}\ a\\
&\mathit{timesM}\ 0\ m && = \mathit{rest}\ 0\\
&\mathit{timesM}\ n\ m && = m \mathbin{:+:} \mathit{timesM}\ (n-1)\ m
\end{aligned}
$$

Details: Note that pattern-matching can be used on numbers. As mentioned earlier, when there is more than one equation that defines a function, the first equation is tried first. If it fails, the second equation is tried, and so on. In the case above, if the first argument to *timesM* is not 0, the first equation will fail. The second equation is then tried, which always succeeds.

So, for example, $\mathit{timesM}\ 3\ b_1$ will repeat the baseline b_1 (to be defined shortly) three times.

To motivate the second auxiliary function, note in Figure 4.1 that there are many melodic lines that consist of a sequence of consecutive notes having the same duration (for example eighth notes in the melody, and dotted

Figure 4.1: Excerpt from Chick Corea's *Children's Songs No. 6*

quarter notes in the bass). To avoid having to write each of these durations explicitly, we will define a function that specifies them just once. To do this, recall that $a\ 4\ qn$ is a concert A quarter note. Then note that, because of currying, $a\ 4$ is a function that can be applied to any duration—i.e. its type is $Dur \rightarrow Music\ a$. In other words, it is a note whose duration has not been specified yet.

With this thought in mind, we can return to the original problem and define a function that takes a duration and a *list* of notes with the aforementioned type, returning a *Music* value with the duration attached to each note appropriately. In Haskell:

```
addDur      :: Dur -> [Dur -> Music a] -> Music a
addDur d ns = let f n = n d
              in line (map f ns)
```

(Compare this idea with Exercise 3.9 in Chapter 3.)

Finally, a function to add a grace note to a note is defined. Grace notes can approach the principal note from above or below; sometimes starting a half-step away, and sometimes a whole step; and having a rhythmic interpretation that is to a large extent up to the performer. In the case of the six uses of grace notes in *Children's Songs No. 6*, we will assume that the grace note begins on the downbeat of the principal note, and thus its duration will subtract from that of the principal note. We will also assume that the grace note duration is 1/8 of that of the principal note. Thus the goal is to define a function:

```
graceNote :: Int -> Music Pitch -> Music Pitch
```

such that $graceNote\ n\ (note\ d\ p)$ is a *Music* value consisting of two notes, the first being the grace note whose duration is $d/8$ and whose pitch is n semitones higher (or lower if n is negative) than p, and the second being the principal note at pitch p but now with duration $7d/8$. In Haskell:

```
graceNote n (Prim (Note d p)) =
  note (d/8) (trans n p) :+: note (7 * d/8) p
graceNote n _                 =
  error "Can only add a grace note to a note."
```

Note that pattern-matching is performed against the nested constructors of *Prim* and *Note*—we cannot match against the application of a function such as *note*. Also note the error message—programs are not expected to ever apply *graceNote* to something other than a single note.

(In Chapter 6 a slightly more general form of *graceNote* will be defined.)

The only special cases that will not be handled using auxiliary functions are the single staccato on note four of bar fifteen, and the single portamento on note three of bar sixteen. These situations will be addressed differently in a later chapter.

4.2.2 Bass Line

With these auxilary functions now defined, the base line in Figure 4.1 can be defined by first noting the three repeating phrases (enclosed in rectangular boxes), which can be captured as follows:

$$\begin{aligned} b_1 &= \mathit{addDur}\ \mathit{dqn}\ [b\ 3,\ \ \mathit{fs}\ 4,\ g\ 4,\ \mathit{fs}\ 4] \\ b_2 &= \mathit{addDur}\ \mathit{dqn}\ [b\ 3,\ \ \mathit{es}\ 4, \mathit{fs}\ 4, \mathit{es}\ 4] \\ b_3 &= \mathit{addDur}\ \mathit{dqn}\ [\mathit{as}\ 3, \mathit{fs}\ 4,\ g\ 4,\ \mathit{fs}\ 4] \end{aligned}$$

Using *timesM* it is then easy to define the entire 28 bars of the base line:

$$\begin{aligned} \mathit{bassLine} = {} & \mathit{timesM}\ 3\ b_1 :+: \mathit{timesM}\ 2\ b_2 :+: \\ & \mathit{timesM}\ 4\ b_3 :+: \mathit{timesM}\ 5\ b_1 \end{aligned}$$

4.2.3 Main Voice

The upper voice of this composition is a bit more tedious to define, but is still straightforward. At the highest level, it consists of the phrase v_1 in the first two bars (in the rectangular box) repeated three times, followed by the remaining melody, which will be named v_2:

$$\mathit{mainVoice} = \mathit{timesM}\ 3\ v_1 :+: v_2$$

The repeating phrase v_1 is defined by:

$$\begin{aligned} v_1 \ \ &= v_{1a} :+: \mathit{graceNote}\ (-1)\ (d\ 5\ \mathit{qn}) :+: v_{1b} \quad \text{-- bars 1-2} \\ v_{1a} &= \mathit{addDur}\ \mathit{en}\ [a\ 5, e\ 5, d\ 5, \mathit{fs}\ 5, \mathit{cs}\ 5, b\ 4, e\ 5, b\ 4] \\ v_{1b} &= \mathit{addDur}\ \mathit{en}\ [\mathit{cs}\ 5, b\ 4] \end{aligned}$$

Note the treatment of the grace note.

The remainder of the main voice, v_2, is defined in seven pieces:

$$v_2 = v_{2a} :+: v_{2b} :+: v_{2c} :+: v_{2d} :+: v_{2e} :+: v_{2f} :+: v_{2g}$$

with each of the pieces defined in Figure 4.2. Note that:

- The phrases are divided so as to (for the most part) line up with bar lines, for convenience. But it may be that this is not the best way to organize the music—for example, we could argue that the last two

```
v2a = line [cs 5 (dhn + dhn), d 5 dhn,
            f 5 hn, gs 5 qn, fs 5 (hn + en), g 5 en]   -- bars 7-11
v2b = addDur en [fs 5, e 5, cs 5, as 4] :+: a 4 dqn :+:
      addDur en [as 4, cs 5, fs 5, e 5, fs 5]          -- bars 12-13
v2c = line [g 5 en, as 5 en, cs 6 (hn + en), d 6 en, cs 6 en] :+:
      e 5 en :+: enr :+:
      line [as 5 en, a 5 en, g 5 en, d 5 qn, c 5 en, cs 5 en]
                                                        -- bars 14-16
v2d = addDur en [fs 5, cs 5, e 5, cs 5,
                 a 4, as 4, d 5, e 5, fs 5]             -- bars 17-18.5
v2e = line [graceNote 2 (e 5 qn), d 5 en, graceNote 2 (d 5 qn), cs 5 en,
            graceNote 1 (cs 5 qn), b 4 (en + hn), cs 5 en, b 4 en]
                                                        -- bars 18.5-20
v2f = line [fs 5 en, a 5 en, b 5 (hn + qn), a 5 en, fs 5 en, e 5 qn,
            d 5 en, fs 5 en, e 5 hn, d 5 hn, fs 5 qn]   -- bars 21-23
v2g = tempo (3/2) (line [cs 5 en, d 5 en, cs 5 en]) :+:
      b 4 (3 * dhn + hn)                                -- bars 24-28
```

Figure 4.2: Bars 7-28

notes in bar 20 form a "pick-up" to the phrase that follows, and thus more logically fall with that following phrase. The organization of the Euterpea code in this way is at the discretion of the composer.

- The stacatto is treated by playing the qurater note as an eighth note; the portamento is ignored. As mentioned earlier, these ornamentations will be addressed differently in a later chapter.
- The triplet of eighth notes in bar 25 is addressed by scaling the tempo by a factor of 3/2.

4.2.4 Putting It All Together

In the Preface to *Children's Songs – 20 Pieces for Keyboard* [Cor94], Chick Corea notes that, "Songs 1 through 15 were composed for the Fender Rhodes." Therefore the MIDI instrument *RhodesPiano* is a logical choice for the transcription of his composition. Furthermore, note in the score that a dotted half-note is specified to have a metronome value of 69. By default, the *play* function in Euterpea uses a tempo equivalent to a quarter note having a metronome value of 120. Therefore the tempo should be scaled by a factor of $(dhn / qn) * (69/120)$.

These two observations lead to the final definition of the transcription of *Children's Songs No. 6* into Euterpea:

```
childSong6 :: Music Pitch
childSong6 = let t = (dhn / qn) * (69/120)
             in instrument RhodesPiano
                           (tempo t (bassLine :=: mainVoice))
```

The intent is that this is the only value that will be of interest to users of this module, and thus *childSong6* is the only name exported from this section of the module, as discussed in Section 4.1.

This example can be played through the command *play childSong6*.

Exercise 4.1 Find a simple piece of music written by your favorite composer, and transcribe it into Euterpea. 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.

4.3 Simple Algorithmic Composition

Algorithmic composition is the process of designing an algorithm (or heuristic) for generating music. There are unlimited possibilites, with some trying to duplicate a particular style of music, others exploring more exotic styles; some based on traditional notions of music theory, others not; some completely deterministic, others probabilistic; and some requiring user interaction, others being completely automatic. Some even are based simply on "interpreting" data—like New York Stock Exchange numbers—in interesting ways! In this textbook a number of algorithmic composition techniques are explored, but the possibilities are endless—hopefully what is presented will motivate the reader to invent new, exciting algorithmic composition techniques.

To give a very tiny glimpse into algorithmic composition, we end this chapter with a very simple example. We will call this example "prefix," for reasons that will become clear shortly.

The user of this algorithm provides an initial melody (or "motif") represented as a list of notes. The main idea is to play every proper (meaning non-empty) prefix of the given melody in succession. So the first thing we do is define a polymorphic function *prefixes* :: [a] → [[a]] that returns all proper prefixes of a list:

$$
\begin{array}{ll}
\mathit{prefixes} & :: [\,a\,] \rightarrow [[\,a\,]] \\
\mathit{prefixes}\ [\,] & = [\,] \\
\mathit{prefixes}\ (x : xs) & = \mathbf{let}\ f\ \mathit{pf} = x : \mathit{pf} \\
 & \ \ \mathbf{in}\ [\,x\,] : \mathit{map}\ f\ (\mathit{prefixes}\ xs)
\end{array}
$$

We can use this to play all prefixes of a given melody *mel* in succession as follows:

$$\mathit{play}\ (\mathit{line}\ (\mathit{concat}\ (\mathit{prefixes}\ \mathit{mel})))$$

But let's do a bit more. Let's create two voices (each using a different instrument), one voice being the reverse of the other, and play them in parallel. And then let's play the whole thing once, then transposed up a perfect fourth (i.e. five semitones), then repeat the whole thing a final time. And, let's package it all into one function:

$$
\begin{array}{l}
\mathit{prefix} :: [\,\mathit{Music}\ a\,] \rightarrow \mathit{Music}\ a \\
\mathit{prefix}\ \mathit{mel} = \mathbf{let}\ m_1 = \mathit{line}\ (\mathit{concat}\ (\mathit{prefixes}\ \mathit{mel})) \\
\qquad\qquad\qquad\quad m_2 = \mathit{transpose}\ 12\ (\mathit{line}\ (\mathit{concat}\ (\mathit{prefixes}\ (\mathit{reverse}\ \mathit{mel})))) \\
\qquad\qquad\qquad\quad m\ \ = \mathit{instrument}\ \mathit{Flute}\ m_1 :=: \mathit{instrument}\ \mathit{VoiceOohs}\ m_2 \\
\qquad\qquad\quad\ \mathbf{in}\ m :+: \mathit{transpose}\ 5\ m :+: m
\end{array}
$$

Here are two melodies (differing only in rhythm) that you can try with this algorithm:

$$
\begin{array}{l}
\mathit{mel}_1 = [\,c\ 5\ \mathit{en}, e\ 5\ \mathit{sn}, g\ 5\ \mathit{en}, b\ 5\ \mathit{sn}, a\ 5\ \mathit{en}, f\ 5\ \mathit{sn}, d\ 5\ \mathit{en}, b\ 4\ \mathit{sn}, c\ 5\ \mathit{en}\,] \\
\mathit{mel}_2 = [\,c\ 5\ \mathit{sn}, e\ 5\ \mathit{sn}, g\ 5\ \mathit{sn}, b\ 5\ \mathit{sn}, a\ 5\ \mathit{sn}, f\ 5\ \mathit{sn}, d\ 5\ \mathit{sn}, b\ 4\ \mathit{sn}, c\ 5\ \mathit{sn}\,]
\end{array}
$$

Although not very sophisticated at all, *prefix* can generate some interesting music from a very small seed.

Another typical approach to algorithmic composition is to specify some constraints on the solution space, and then generate lots of solutions that satisfy those constraints. The user can then choose one of the solutions based on aesthetic preferences.

As a simple example of this, how do we choose the original melody in the prefix program above? We could require that all solutions be a multiple of some preferred meter. For example, in triple meter (say, 3/4 time) we might wish for the solutions to be multiples of 3 quarter-note beats (i.e.

one measure), or in $4/4$ time, multiples of 4 beats. In this way the result is always an integer number of measures. If the original melody consists of notes all of the same duration, say one beat, then the prefixes, when played sequentially, will have a total duration that is the sum of the numbers 1 through n, where n is the length of melody in beats. That sum is $n*(n+1)/2$. The first ten sums in this series are:

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...$$

The second, third, fifth, sixth, eighth, and ninth of these are divisible by 3, and the seventh and eighth are divisible by 4. When rendering the result we could then, for exaple, place an accent on the first note in each of these implied measures, thus giving the result more of a musical feel. (Placing an accent on a note will be explained in Chapters 6 and **??**.)

Exercise 4.2 Try using *prefix* on your own melodies. Indeed, note that the list of notes could in general be a list of any *Music* values.

Exercise 4.3 Try making the following changes to *prefix*:

1. Use different instruments.
2. Change the definition of m in some way.
3. Compose the result in a different way.

Chapter 5

Syntactic Magic

This chapter introduces several more of Haskell's syntactic devices that faciliate writing concise and intuitive programs. These devices will be used frequently in the remainder of the text.

5.1 Sections

The use of currying was introduced in Chapter 3 as a way to simplify programs. This is a syntactic device that relies on the way that normal functions are applied, and how those applications are parsed.

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 $(x + y)$, we 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:

$$
\begin{aligned}
f_1\ y \quad &= x + y \\
f_2\ x \quad &= x + y \\
f_3\ x\ y &= x + y
\end{aligned}
$$

For example, suppose we wish to remove all absolute pitches greater than 99 from a list, perhaps because everything above that value is assumed to be unplayable. There is a pre-defined function in Haskell that can help to achieve this:

$$\mathit{filter} :: (a \to \mathit{Bool}) \to [a] \to [a]$$

filter $p\ xs$ returns a list for which each element x satisfies the predicate p; i.e. $p\ x$ is *True*.

Using *filter*, we can then write:

$$\begin{array}{l}\mathit{playable} \quad :: [\mathit{AbsPitch}] \to [\mathit{AbsPitch}] \\ \mathit{playable}\ \mathit{xs} = \mathbf{let}\ \mathit{test}\ \mathit{ap} = \mathit{ap} < 100 \\ \qquad\qquad\quad\ \ \mathbf{in}\ \mathit{filter}\ \mathit{test}\ \mathit{xs}\end{array}$$

But using a section, we can write this more succinctly as:

$$\begin{array}{l}\mathit{playable} \quad :: [\mathit{AbsPitch}] \to [\mathit{AbsPitch}] \\ \mathit{playable}\ \mathit{xs} = \mathit{filter}\ (<100)\ \mathit{xs}\end{array}$$

which can be further simplified using currying:

$$\begin{array}{l}\mathit{playable} :: [\mathit{AbsPitch}] \to [\mathit{AbsPitch}] \\ \mathit{playable} = \mathit{filter}\ (<100)\end{array}$$

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. Many more examples of this kind of reasoning will appear throughout the text.

Exercise 5.1 Define a function *twice* that, given a function f, returns a function that applies f twice to its argument. For example:

$$(\mathit{twice}\ (+1))\ 2 \Rightarrow 4$$

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

Exercise 5.2 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:

$$\mathit{power}\ (+2)\ 5\ 1 \Longrightarrow 11$$

Use *power* in a musical context to define something useful.

5.2 Anonymous Functions

Another way to define a function in Haskell is in some sense the most fundamental: it is called an *anonymous function*, or *lambda expression* (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 do not wish to name them, which is why they are called "anonymous." Anonymity is a property also shared by sections, but sections can only be derived from an existing infix operator.

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

As another example, to raise the pitch of every element in a list of pitches *ps* by an octave, we could write:

$$map\ (\lambda p \to pitch\ (absPitch\ p + 12))\ ps$$

An even better example is an anonymous function that pattern-matches its argument, as in the following, which doubles the duration of every note in a list of notes *ns*:

$$map\ (\lambda(Note\ d\ p) \to Note\ (2 * d)\ p)\ ns$$

> **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:

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

Details: $\lambda x\ y\ z \rightarrow exp$ is shorthand for $\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow exp$.

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

$$\begin{aligned}
(x+) &\Rightarrow \lambda y \rightarrow x + y \\
(+y) &\Rightarrow \lambda x \rightarrow x + y \\
(+) &\Rightarrow \lambda x\ y \rightarrow x + y
\end{aligned}$$

Exercise 5.3 Suppose we define a function *fix* as:

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

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

$$\begin{aligned}
&remainder &&:: Integer \rightarrow Integer \rightarrow Integer \\
&remainder\ a\ b &&= \mathbf{if}\ a < b\ \mathbf{then}\ a \\
& &&\ \mathbf{else}\ remainder\ (a - b)\ b
\end{aligned}$$

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?

5.3 List Comprehensions

Haskell has a convenient and intuitive way to define a list in such a way that it resembles the definition of a *set* in mathematics. For example, recall in the last chapter the definition of the function *addDur*:

$$\begin{aligned}
&addDur &&:: Dur \rightarrow [Dur \rightarrow Music\ a] \rightarrow Music\ a \\
&addDur\ d\ ns &&= \mathbf{let}\ f\ n = n\ d \\
& &&\ \mathbf{in}\ line\ (map\ f\ ns)
\end{aligned}$$

Here *ns* is a list of notes, each of which does not have a duration yet assigned to it. If we think of this as a set, we might be led to write the following solution in mathematical notation:

$\{n\ d \mid n \in ns\}$

which can be read, "the set of all notes $n\ d$ such that n is an element of ns." Indeed, using a Haskell *list comprehension* we can write almost exactly the same thing:

$$[n\ d \mid n \leftarrow ns]$$

The difference, of course, is that the above expression generates an (ordered) list in Haskell, not an (unordered) set in mathematics.

List comprehensions allow us to rewrite the definition of *addDur* much more succinctly and elegantly:

$$\begin{aligned}&addDur && :: Dur \rightarrow [Dur \rightarrow Music\ a] \rightarrow Music\ a\\ &addDur\ d\ ns && = line\ [n\ d \mid n \leftarrow ns]\end{aligned}$$

Details: Liberty is again taken in type-setting by using the symbol $\leftarrow$ to mean "is an element of." When writing your programs, you will have to type "<-" instead.

The expression $[exp \mid x \leftarrow xs]$ is actually shorthand for the expression $map\ (\lambda x \rightarrow exp)\ xs$. The form $x \leftarrow xs$ is called a *generator*, and in general more than one is allowed, as in:

$$[(x, y) \mid x \leftarrow [0, 1, 2], y \leftarrow [\texttt{'a'}, \texttt{'b'}]]$$

which evaluates to the list:

$$[(0, \texttt{'a'}), (0, \texttt{'b'}), (1, \texttt{'a'}), (1, \texttt{'b'}), (2, \texttt{'a'}), (2, \texttt{'b'})]$$

The order here is important; that is, note that the left-most generator changes least quickly.

It is also possible to *filter* values as they are generated; for example, we can modify the above example to eliminate the odd integers in the first list:

$$[(x, y) \mid x \leftarrow [0, 1, 2], even\ x, y \leftarrow [\texttt{'a'}, \texttt{'b'}]]$$

where $even\ n$ returns $True$ if n is even. This example evaluates to:

$$[(0, \texttt{'a'}), (0, \texttt{'b'}), (2, \texttt{'a'}), (2, \texttt{'b'})]$$

Details: When reasoning about list comprehensions (e.g. when doing proof by calculation), we can use the following syntactic translation into pure functions:

$$
\begin{array}{ll}
[\,e \mid \mathit{True}\,] & = [\,e\,] \\
[\,e \mid q\,] & = [\,e \mid q, \mathit{True}\,] \\
[\,e \mid b, qs\,] & = \textbf{if}\ b\ \textbf{then}\ [\,e \mid qs\,]\ \textbf{else}\ [\,] \\
[\,e \mid p \leftarrow xs, qs\,] & = \textbf{let}\ ok\ p = [\,e \mid qs\,] \\
 & \qquad\quad ok\ \_ = [\,] \\
 & \quad\ \textbf{in}\ concatMap\ ok\ xs \\
[\,e \mid \textbf{let}\ decls, qs\,] & = \textbf{let}\ decls\ \textbf{in}\ [\,e \mid qs\,]
\end{array}
$$

where q is a single qualifier, qs is a sequence of qualifiers, b is a Boolean, p is a pattern, and $decls$ is a sequence of variable bindings (a feature of list comprehensions not explained earlier).

5.3.1 Arithmetic Sequences

Another convenient syntax for lists whose elements can be enumerated is called an *arithmetic sequence*. For example, the arithmetic sequence $[1\,..\,10]$ is equivalent to the list:

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

There are actually four different versions of arithmetic sequences, some of which generate *infinite* lists (whose use will be discussed in a later chapter). In the following, let $a = n' - n$:

$$
\begin{array}{ll}
[\,n\,..\,] & \text{-- infinite list } n,\ n+1,\ n+2,\ ... \\
[\,n, n'\,..\,] & \text{-- infinite list } n,\ n+a,\ n+2*a,\ ... \\
[\,n\,..\,m\,] & \text{-- finite list } n,\ n+1,\ n+2,\ ...,\ m \\
[\,n, n'\,..\,m\,] & \text{-- finite list } n,\ n+a,\ n+2*a,\ ...,\ m
\end{array}
$$

Arithmetic sequences are discussed in greater detail in Appendix B.

Exercise 5.4 Using list comprehensions, define a function:

$$apPairs :: [AbsPitch] \rightarrow [AbsPitch] \rightarrow [(AbsPitch, AbsPitch)]$$

such that $apPairs\ aps_1\ aps_2$ is a list of all combinations of the absolute pitches in aps_1 and aps_2. Furthermore, for each pair (ap_1, ap_2) in the result, the absolute value of $ap_1 - ap_2$ must be greater than two and less than eight.

Finally, write a function to turn the result of $apPairs$ into a $Music\ Pitch$ value by playing each pair of pitches in parallel, and stringing them all together sequentially. Try varying the rhythm by, for example, using an

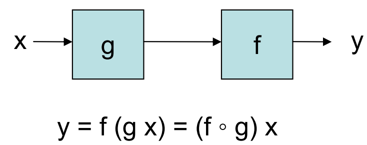


Figure 5.1: Gluing Two Functions Together

eighth note when the first absolute pitch is odd, and a sixteenth note when it is even, or some other criterion.

Test your functions by using arithemtic sequences to generate the two lists of arguments given to *apPairs*.

5.4 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* (just as in mathematics), and Haskell pre-defines a simple infix operator $(\circ)$ to achieve it, as follows:

$$
\begin{array}{ll}
(\circ) & :: (b \to c) \to (a \to b) \to a \to c \\
(f \circ g)\ x = f\ (g\ x) &
\end{array}
$$

Details: The symbol for function composition is typeset in this textbook as $\circ$, which is consistent with 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 we 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 5.1.

The ability to compose functions using $(\circ)$ is quite handy. For example, recall the last version of *hList*:

$$hList\ d\ ps = line\ (map\ (hNote\ d)\ ps)$$

We can do two simplifications here. First, rewrite the right-hand side using function composition:

$$hList\ d\ ps = (line \circ map\ (hNote\ d))\ ps$$

Then, use currying simplification:

$$hList\ d = line \circ map\ (hNote\ d)$$

5.5 Higher-Order Thinking

It is worth taking a deep breath here and contemplating what has been done with *hList*, which has gone through quite a few transformations. Here is the original definition given in Chapter 1:

$$\begin{aligned} &hList\ d\ [\,] &&= rest\ 0 \\ &hList\ d\ (p : ps) &&= hNote\ d\ p :+: hList\ d\ ps \end{aligned}$$

Compare this to the definition above. You may be distressed to think that you have to go through all of these transformations just to write a relatively simple function! There are two points to make about this: First, you do not have to make *any* of these transformations if you do not want to. All of these versions of *hList* are correct, and they all run about equally fast. They are explained here for pedagogical purposes, so that you understand the full power of Haskell. Second, with practice, you will find that you can write the concise higher-order versions of many functions straight away, without going through all of the steps presented here.

As mentioned earlier, one thing that helps is to start *thinking* in "higher-order" terms. To facilitate this way of thinking it is helpful to write type signatures that reflect more closely their higher-order nature. For example, recall these type signatures for *map*, *filter*, and $(\circ)$:

$$\begin{aligned} &map &&:: (a \to b) \to [\,a\,] \to [\,b\,] \\ &filter &&:: (a \to Bool) \to [\,a\,] \to [\,a\,] \\ &(\circ) &&:: (b \to c) \to (a \to b) \to a \to c \end{aligned}$$

Also recall that the arrow in function types is right associative. Therefore, another completely equivalent way to write the above type signatures is:

$$\begin{aligned} &map &&:: (a \to b) \to ([\,a\,] \to [\,b\,]) \\ &filter &&:: (a \to Bool) \to ([\,a\,] \to [\,a\,]) \\ &(\circ) &&:: (b \to c) \to (a \to b) \to (a \to c) \end{aligned}$$

Although equivalent, the latter versions emphasize the fact that each of these functions returns a function as its result. *map* essentially "lifts" a function on elements to a function on lists of elements. *filter* converts a predicate into a function on lists. And $(\circ)$ returns a function that is the composition of its two functional arguments.

So for example, using higher-order thinking, *map* $(+12)$ is a function that transposes a list of absolute pitches by one octave. *filter* (<100) is a function that removes all absolute pitches greater than or equal to 100 (as discussed earlier). And therefore *map* $(+12) \circ$ *filter* (<100) first does the filtering, and then does the transposition. All very consise and very natural using higher-order thinking.

In the remainder of this textbook definitions such as this will be written directly, using a small set of rich polymorphic functions such as *foldl*, *map*, *filter*, $(\circ)$, and a few other functions drawn from the Standard Prelude and other standard libraries.

5.6 Infix Function Application

Haskell predefines an infix operator to apply a function to a value:

$$f \; \$ \; x = f \; x$$

At first glance this does not seem very useful—after all, why not simply write $f \; x$ instead of $f \; \$ \; x$?

But in fact this operator has a very useful purpose: eliminating parentheses! In the Standard Prelude, (\$) is defined to be right associative, and to have the lowest precedence level, via the fixity declaration:

infixr 0 \$

Therefore, note that $f \; (g \; x)$ is the same as $f \; \$ \; g \; x$ (remember that normal function application always has higher precedence than infix operator application), and $f \; (x + 1)$ is the same as $f \; \$ \; x + 1$. This "trick" is especially useful when there is a sequence of nested, parenthesized expresssions. For example, recall the following definition from the last chapter:

$$
\begin{aligned}
&\mathit{childSong6} = \textbf{let}\ t = (\mathit{dhn} / \mathit{qn}) * (69/120) \\
&\qquad\qquad\quad \textbf{in}\ \mathit{instrument}\ \mathit{RhodesPiano} \\
&\qquad\qquad\qquad\qquad (\mathit{tempo}\ t\ (\mathit{bassLine} :=: \mathit{mainVoice}))
\end{aligned}
$$

We can rewrite the last few lines a bit more clearly as follows:

$$\begin{array}{l} childSong6 = \textbf{let}\ t = (dhn\,/\,qn) * (69\,/\,120) \\ \qquad\qquad\quad \textbf{in}\ instrument\ RhodesPiano\ \$ \\ \qquad\qquad\qquad tempo\ t\ \$ \\ \qquad\qquad\qquad bassLine :=: mainVoice \end{array}$$

Or, on a single line, instead of:

$$instrument\ RhodesPiano\ (tempo\ t\ (bassLine :=: mainVoice))$$

we can write:

$$instrument\ RhodesPiano\ \$\ tempo\ t\ \$\ bassLine :=: mainVoice$$

Exercise 5.5 The last definition of *hList* still has an argument d on the left-hand side, and one occurence of d on the right-hand side. Is there some way to eliminate it using currying simplification? (Hint: the answer is yes, but the solution is a bit perverse, and is not recommended as a way to write your code!)

Exercise 5.6 Use *line*, *map* and (\$) to give a concise definition of *addDur*.

Exercise 5.7 Rewrite this example:

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

using a composition of sections.

Exercise 5.8 Consider the expression:

$$map\ f\ (map\ g\ xs)$$

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

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

as a "map of a map" (i.e. using two maps).

Exercise 5.9 Go back to any exercises prior to this chapter, and simplify your solutions using ideas learned here.

Exercise 5.10 Using higher-order functions introduced in this chapter, fill in the two missing functions, f_1 and f_2, in the evaluation below so that it is valid:

$$f_1\ (f_2\ (*)\ [1, 2, 3, 4])\ 5 \Rightarrow [5, 10, 15, 20]$$

Chapter 6

More Music

module *Euterpea.Music.Note.MoreMusic* **where**
import *Euterpea.Music.Note.Music*

This chapter explores a number of simple musical ideas, and contributes to a growing collection of Euterpea functions for expressing those ideas.

6.1 Delay and Repeat

We can delay the start of a music value simply by inserting a rest in front of it, which can be packaged in a function as follows:

$$\begin{array}{ll} \mathit{delayM} & :: \mathit{Dur} \to \mathit{Music}\ a \to \mathit{Music}\ a \\ \mathit{delayM}\ d\ m & = \mathit{rest}\ d :\!+\!: m \end{array}$$

With *delayM* it is easy to write canon-like structures such as $m{:=:}\mathit{delayM}\ d\ m$, a song written in rounds (see Exercise 3.14), and so on.

Recall from Chapter 4 the function *timesM* that repeats a musical phrase a certain number of times:

$$\begin{array}{ll} \mathit{timesM} & :: \mathit{Int} \to \mathit{Music}\ a \to \mathit{Music}\ a \\ \mathit{timesM}\ 0\ m & = \mathit{rest}\ 0 \\ \mathit{timesM}\ n\ m & = m :\!+\!: \mathit{timesM}\ (n-1)\ m \end{array}$$

More interestingly, Haskell's non-strict semantics allows us to define *infinite* musical values. For example, a musical value may be repeated *ad nauseam* 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. Functions that create such contexts will be described shortly.

6.2 Inversion and Retrograde

The notions of inversion, retrograde, retrograde inversion, etc. as used in twelve-tone theory are also easily captured in Euterpea. These terms are usually applied only to a "line" 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 pitch *p* is converted into an absolute pitch $ap - (absPitch\ p - ap)$, in other words $2 * ap - absPitch\ p$.

To do all this in Haskell, a transformation from a line created by *line* to a list is defined:

```
lineToList                    :: Music a -> [Music a]
lineToList (Prim (Rest 0)) = []
lineToList (n :+: ns)         = n : lineToList ns
lineToList _                  =
  error "lineToList: argument not created by function line"
```

Using this function it is then straightforward to define *invert*:

```
invert :: Music Pitch -> Music Pitch
invert m =
  let l@(Prim (Note _ r) : _) = lineToList m
      inv (Prim (Note d p)) =
                  note d (pitch (2 * absPitch r - absPitch p))
      inv (Prim (Rest d))    = rest d
  in line (map inv l)
```

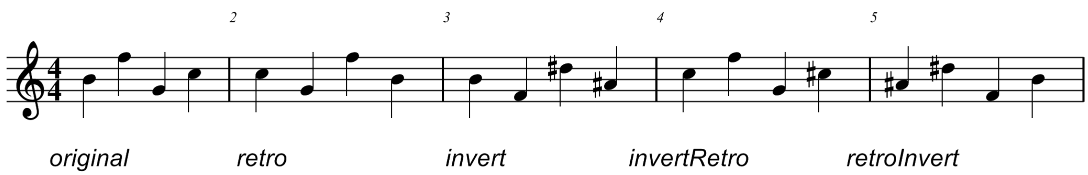


Figure 6.1: A Simple Melody and Four Transformations

Details: The pattern $l@(Prim\ (Note\ \_\ r) : \_)$ is called an *as pattern*. It behaves just like the pattern $Prim\ (Note\ \_\ r) : \_$ but additionally binds l to the value of a successful match to that pattern. l can then be used wherever it is in scope, such as in the last line of the function definition.

With *lineToList* and *invert* it is then easy to define the remaining functions via composition:

$$
\begin{array}{lcl}
\mathit{retro}, \mathit{retroInvert}, \mathit{invertRetro} :: \mathit{Music\ Pitch} \rightarrow \mathit{Music\ Pitch} \\
\mathit{retro} & = & \mathit{line} \circ \mathit{reverse} \circ \mathit{lineToList} \\
\mathit{retroInvert} & = & \mathit{retro} \circ \mathit{invert} \\
\mathit{invertRetro} & = & \mathit{invert} \circ \mathit{retro}
\end{array}
$$

As an example of these concepts, Figure 6.1 shows a simple melody (not a complete twelve-tone row) and four transformations of it.

Exercise 6.1 Show that *retro* ∘ *retro*, *invert* ∘ *invert*, and *retroInvert* ∘ *invertRetro* are the identity on values created by *line*. (You may use the lemma that $reverse\ (reverse\ l) = l$.)

Exercise 6.2 Define a function *properRow* :: *Music Pitch* → *Bool* that determines whether or not its argument is a "proper" twelve-tone row, meaning that: (a) it must have exactly twelve notes, and (b) each unique pitch class is used exactly once (regardless of the octave). Enharmonically equivalent pitch classes are *not* considered unique. You may assume that the *Music Pitch* value is generated by the function *line*, but note that rests are allowed.

Exercise 6.3 Define a function *palin* :: *Music Pitch* → *Bool* that determines whether or not a given line (as generated by the *line* function) is a palin-

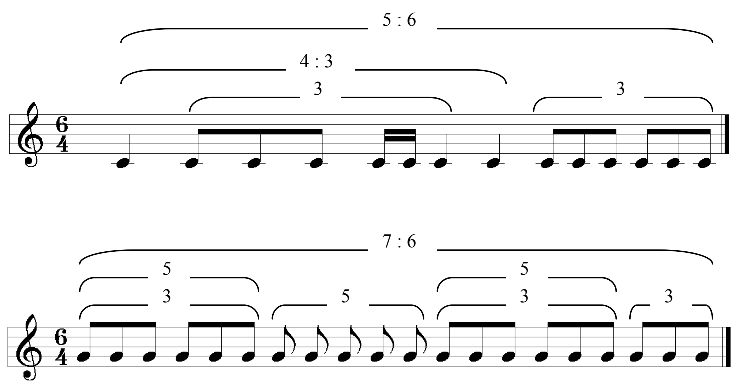


Figure 6.2: Nested Polyrhythms (top: pr_1; bottom: pr_2)

drome or not. You should ignore rests, and disregard note durations—the main question is whether or not the melody is a palindrome.

Exercise 6.4 Define a function *retroPitches* :: *Music Pitch* → *Music Pitch* that reverses the pitches in a line, but maintains the durations in the same order from beginning to end. For example:

```
retroPitches (line [c 4 en, d 4 qn])
⟹ (line [d 4 en, c 4 qn])
```

6.3 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 (recall that this idea was used in Chapter 4). In fact *tempo* can be used to create quite complex rhythmical patterns. For example, consider the "nested polyrhythms" shown in Figure 6.2. They can be expressed naturally in Euterpea as follows (note the use of a **let** clause in pr_2 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) :+:
```

$$\begin{array}{l} \qquad\qquad mkLn\ 1\ p\ qn) \mathbin{:+:} \\ tempo\ (3/2)\ (mkLn\ 6\ p\ en)) \end{array}$$

$$
\begin{array}{l}
pr_2\ p = \\
\quad \textbf{let}\ m_1 = tempo\ (5/4)\ (tempo\ (3/2)\ m_2 :+: m_2) \\
\qquad m_2 = mkLn\ 3\ p\ en \\
\quad \textbf{in}\ tempo\ (7/6)\ (m_1 :+: \\
\qquad\qquad\qquad tempo\ (5/4)\ (mkLn\ 5\ p\ en) :+: \\
\qquad\qquad\qquad m_1 :+: \\
\qquad\qquad\qquad tempo\ (3/2)\ m_2) \\
mkLn :: Int \to p \to Dur \to Music\ p \\
mkLn\ n\ p\ d = line\ \$\ take\ n\ \$\ repeat\ \$\ note\ d\ p
\end{array}
$$

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 pr_1 and pr_2 in parallel using middle C and middle G, respectively, we can write:

$$
\begin{array}{l}
pr_{12} :: Music\ Pitch \\
pr_{12} = pr_1\ (C, 4) :=: pr_2\ (G, 4)
\end{array}
$$

6.4 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:

$$
\begin{array}{ll}
(=:=) & :: Dur \to Dur \to Music\ a \to Music\ a \\
old =:= new = tempo\ (new / old) &
\end{array}
$$

Of course, using the new function is not much shorter than using *tempo* directly, but it may have mnemonic value.

6.5 Computing Duration

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

$$
\begin{array}{ll}
dur & :: Music\ a \rightarrow Dur \\
dur\ (Prim\ (Note\ d\ \_)) & = d \\
dur\ (Prim\ (Rest\ d)) & = d \\
dur\ (m_1 :+: m_2) & = dur\ m_1 + dur\ m_2 \\
dur\ (m_1 :=: m_2) & = dur\ m_1\ \text{`}max\text{`}\ dur\ m_2 \\
dur\ (Modify\ (Tempo\ r)\ m) & = dur\ m / r \\
dur\ (Modify\ \_\ m) & = dur\ m
\end{array}
$$

The duration of a primitive value is obvious. The duration of $m_1 :+: m_2$ is the sum of the two, and the duration of $m_1 :=: m_2$ is the maximum of the two. The only tricky case is the duration of a music value that is modified by the *Tempo* atttribute—in this case the duration must be scaled appropriately.

Note that the duration of a music value that is conceptually infinite in duration will be $\perp$, since *dur* will not terminate. (Similary, taking the length of an infinite list is $\perp$.) For example:

$$
\begin{array}{l}
dur\ (repeatM\ (a\ 4\ qn)) \\
\Rightarrow dur\ (a\ 4\ qn :+: repeatM\ (a\ 4\ qn)) \\
\Rightarrow dur\ (a\ 4\ qn) + dur\ (repeatM\ (a\ 4\ qn)) \\
\Rightarrow qn + dur\ (repeatM\ (a\ 4\ qn)) \\
\Rightarrow qn + qn + dur\ (repeatM\ (a\ 4\ qn)) \\
\Rightarrow ... \\
\Rightarrow \perp
\end{array}
$$

6.6 Super-retrograde

Using *dur* we can define a function *revM* that reverses any *Music* value whose duration is finite (and is thus considerably more useful than *retro* defined earlier):

$$
\begin{array}{l}
\mathit{revM} \quad :: \mathit{Music}\ a \to \mathit{Music}\ a \\
\mathit{revM}\ n@(\mathit{Prim}\ \_) = n \\
\mathit{revM}\ (\mathit{Modify}\ c\ m) = \mathit{Modify}\ c\ (\mathit{revM}\ m) \\
\mathit{revM}\ (m_1 :+: m_2) = \mathit{revM}\ m_2 :+: \mathit{revM}\ m_1 \\
\mathit{revM}\ (m_1 :=: m_2) = \\
\quad \textbf{let}\ d_1 = \mathit{dur}\ m_1 \\
\quad \phantom{\textbf{let}}\ d_2 = \mathit{dur}\ m_2 \\
\quad \textbf{in if}\ d_1 > d_2\ \textbf{then}\ \mathit{revM}\ m_1 :=: (\mathit{rest}\ (d_1 - d_2) :+: \mathit{revM}\ m_2) \\
\quad \phantom{\textbf{in if}\ d_1 > d_2}\ \textbf{else}\ (\mathit{rest}\ (d_2 - d_1) :+: \mathit{revM}\ m_1) :=: \mathit{revM}\ m_2
\end{array}
$$

The first three cases are easy, but the last case is a bit tricky. The parallel constructor (:=:) implicitly begins each of its music values at the same time. But if one is shorter than the other, then, when reversed, a *rest* must be inserted before the shorter one, to account for the difference.

Note that *revM* of a *Music* value whose duration is infinite is $\perp$. (Analogously, reversing an infinite list is $\perp$.)

6.7 *takeM* and *dropM*

Two other useful operations on *Music* values is the ability to "take" the first so many beats (in whole notes), discarding the rest, and conversely, the ability to "drop" the first so many beats, returning what is left. We will first define a function $\mathit{takeM} :: \mathit{Dur} \to \mathit{Music}\ a \to \mathit{Music}\ a$ such that $\mathit{takeM}\ d\ m$ is a *prefix* of m having duration d. In other words, it "takes" only the first d beats (in whole notes) of m. We can define this function as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{takeM} :: \mathit{Dur} \to \mathit{Music}\ a \to \mathit{Music}\ a} \\
\mathit{takeM}\ d\ m \mid d \leqslant 0 & = \mathit{rest}\ 0 \\
\mathit{takeM}\ d\ (\mathit{Prim}\ (\mathit{Note}\ \mathit{oldD}\ p)) & = \mathit{note}\ (\mathit{min}\ \mathit{oldD}\ d)\ p \\
\mathit{takeM}\ d\ (\mathit{Prim}\ (\mathit{Rest}\ \mathit{oldD})) & = \mathit{rest}\ (\mathit{min}\ \mathit{oldD}\ d) \\
\mathit{takeM}\ d\ (m_1 :=: m_2) & = \mathit{takeM}\ d\ m_1 :=: \mathit{takeM}\ d\ m_2 \\
\mathit{takeM}\ d\ (m_1 :+: m_2) & = \textbf{let}\ m'_1 = \mathit{takeM}\ d\ m_1 \\
 & \phantom{= \textbf{let}}\ m'_2 = \mathit{takeM}\ (d - \mathit{dur}\ m'_1)\ m_2 \\
 & \ \textbf{in}\ m'_1 :+: m'_2 \\
\mathit{takeM}\ d\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ m) & = \mathit{tempo}\ r\ (\mathit{takeM}\ (d * r)\ m) \\
\mathit{takeM}\ d\ (\mathit{Modify}\ c\ m) & = \mathit{Modify}\ c\ (\mathit{takeM}\ d\ m)
\end{array}
$$

This definition is fairly straightforward, except for the case of sequential composition, where two cases arise: (1) if d is greater than $\mathit{dur}\ m_1$, then we return *all* of m_1 (i.e. $m'_1 = m_1$), followed by $d - \mathit{dur}\ m'_1$ beats of m_2, and (2) if d is less than $dur\ m_1$, then we return d beats of m_1 (i.e. m'_1), followed by nothing (since $d - dur\ m'_1$ will be zero). Note that this strategy will work even if m_1 or m_2 is infinite.

Similarly, we can define a function $dropM :: Dur \rightarrow Music\ a \rightarrow Music\ a$ such that $dropM\ \ d\ \ m$ is a *suffix* of m where the first d beats (in whole notes) of m have been "dropped:"

$$
\begin{array}{ll}
\multicolumn{2}{l}{dropM :: Dur \rightarrow Music\ a \rightarrow Music\ a} \\
dropM\ \ d\ m \mid d \leqslant 0 & = m \\
dropM\ \ d\ (Prim\ (Note\ oldD\ p)) & = note\ (max\ (oldD - d)\ 0)\ p \\
dropM\ \ d\ (Prim\ (Rest\ oldD)) & = rest\ (max\ (oldD - d)\ 0) \\
dropM\ \ d\ (m_1 :=: m_2) & = dropM\ \ d\ m_1 :=: dropM\ \ d\ m_2 \\
dropM\ \ d\ (m_1 :+: m_2) & = \mathbf{let}\ m'_1 = dropM\ \ d\ m_1 \\
 & \qquad\quad m'_2 = dropM\ \ (d - dur\ m_1)\ m_2 \\
 & \quad\ \mathbf{in}\ m'_1 :+: m'_2 \\
dropM\ \ d\ (Modify\ (Tempo\ r)\ m) & = tempo\ r\ (dropM\ \ (d * r)\ m) \\
dropM\ \ d\ (Modify\ c\ m) & = Modify\ c\ (dropM\ \ d\ m)
\end{array}
$$

This definition is also straightforward, except for the case of sequential composition. Again, two cases arise: (1) if d is greater than $dur\ m_1$, then we drop m_1 altogether (i.e. m'_1 will be $rest\ 0$), and simply drop $d - dur\ m_1$ from m_2, and (2) if d is less than $dur\ m_1$, then we return m'_1 followed by all of m_2 (since $d - dur\ m_1$ will be negative). This definition too will work for infinite values of m_1 or m_2.

6.8 Removing Zeros

Note that functions such as *timesM*, *line*, *revM*, *takeM* and *dropM* occasionally insert rests of zero duration, and in the case of *takeM* and *dropM*, may insert notes of zero duration. Doing this makes the code simpler and more elegant, and since we cannot hear the effect of the zero-duration events, the musical result is the same.

On the other hand, these extraneous notes and rests (which we will call "zeros") can be annoying when viewing the textual (rather than audible) representation of the result. To alleviate this problem, we define a function that removes them from a given *Music* value:

$$
\begin{array}{l}
\mathit{removeZeros} :: \mathit{Music}\ a \rightarrow \mathit{Music}\ a \\
\mathit{removeZeros}\ (\mathit{Prim}\ p) \qquad = \mathit{Prim}\ p \\
\mathit{removeZeros}\ (m_1 :+: m_2) \quad = \\
\quad \textbf{let}\ m'_1 = \mathit{removeZeros}\ m_1 \\
\qquad\quad m'_2 = \mathit{removeZeros}\ m_2 \\
\quad \textbf{in case}\ (m'_1, m'_2)\ \textbf{of} \\
\qquad (\mathit{Prim}\ (\mathit{Note}\ 0\ p), m) \rightarrow m \\
\qquad (\mathit{Prim}\ (\mathit{Rest}\ 0), m) \quad \rightarrow m \\
\qquad (m, \mathit{Prim}\ (\mathit{Note}\ 0\ p)) \rightarrow m \\
\qquad (m, \mathit{Prim}\ (\mathit{Rest}\ 0)) \quad \rightarrow m \\
\qquad (m_1, m_2) \qquad\qquad\quad \rightarrow m_1 :+: m_2 \\
\mathit{removeZeros}\ (m_1 :=: m_2) \quad = \\
\quad \textbf{let}\ m'_1 = \mathit{removeZeros}\ m_1 \\
\qquad\quad m'_2 = \mathit{removeZeros}\ m_2 \\
\quad \textbf{in case}\ (m'_1, m'_2)\ \textbf{of} \\
\qquad (\mathit{Prim}\ (\mathit{Note}\ 0\ p), m) \rightarrow m \\
\qquad (\mathit{Prim}\ (\mathit{Rest}\ 0), m) \quad \rightarrow m \\
\qquad (m, \mathit{Prim}\ (\mathit{Note}\ 0\ p)) \rightarrow m \\
\qquad (m, \mathit{Prim}\ (\mathit{Rest}\ 0)) \quad \rightarrow m \\
\qquad (m_1, m_2) \qquad\qquad\quad \rightarrow m_1 :=: m_2 \\
\mathit{removeZeros}\ (\mathit{Modify}\ c\ m) = \mathit{Modify}\ c\ (\mathit{removeZeros}\ m)
\end{array}
$$

Details: A **case** expression can only match against one value. To match against more than one value, we can place them in a tuple of the appropriate length. In the case above, *removeZeros* matches against m'_1 and m'_2 by placing them in a pair (m'_1, m'_2).

This function depends on the "musical axioms" that if m_1 in either $m_1 :+: m_2$ or $m_1 :=: m_2$ is a zero, then the latter expressions are equivalent to just m_2. Similarly, if m_2 is a zero, they are equivalent to just m_1. Although intuitive, a formal proof of these axioms is deferred until Chapter 11.

As an example of using *removeZeros*, consider the *Music* value:

$$m = c\ 4\ en :+: \mathit{repeatM}\ (d\ 4\ en)$$

Then note that:

$$
\begin{array}{l}
\mathit{takeM}\ \mathit{hn}\ (\mathit{dropM}\ \mathit{hn}\ m)\\
\Longrightarrow\\
\mathit{Prim}\ (\mathit{Note}\ (0\,\%\,1)\ (C,4))\ {:+:}\ (\mathit{Prim}\ (\mathit{Note}\ (0\,\%\,1)\ (D,4))\ {:+:}\\
(\mathit{Prim}\ (\mathit{Note}\ (0\,\%\,1)\ (D,4))\ {:+:}\ (\mathit{Prim}\ (\mathit{Note}\ (0\,\%\,1)\ (D,4))\ {:+:}\\
(\mathit{Prim}\ (\mathit{Note}\ (1\,\%\,8)\ (D,4))\ {:+:}\ (\mathit{Prim}\ (\mathit{Note}\ (1\,\%\,8)\ (D,4))\ {:+:}\\
(\mathit{Prim}\ (\mathit{Note}\ (1\,\%\,8)\ (D,4))\ {:+:}\ (\mathit{Prim}\ (\mathit{Note}\ (1\,\%\,8)\ (D,4))\ {:+:}\\
\mathit{Prim}\ (\mathit{Rest}\ (0\,\%\,1)))))))))
\end{array}
$$

Note the zero-duration notes and rests. But if we apply *removeZeros* to the result we get:

$$
\begin{array}{l}
\mathit{removeZeros}\ (\mathit{takeM}\ \mathit{hn}\ (\mathit{dropM}\ \mathit{hn}\ m))\\
\Longrightarrow\\
\mathit{Prim}\ (\mathit{Note}\ (1\,\%\,8)\ (D,4))\ {:+:}\ (\mathit{Prim}\ (\mathit{Note}\ (1\,\%\,8)\ (D,4))\ {:+:}\\
(\mathit{Prim}\ (\mathit{Note}\ (1\,\%\,8)\ (D,4))\ {:+:}\ \mathit{Prim}\ (\mathit{Note}\ (1\,\%\,8)\ (D,4))))
\end{array}
$$

Both the zero-duration rests and notes have been removed.

6.9 Truncating Parallel Composition

The duration of $m_1 :=: m_2$ is the maximum of the durations of m_1 and m_2 (and thus if one is infinite, so is the result). However, sometimes it is useful to have the result be of duration equal to the *shorter* of the two. Defining a function to achieve this is not as easy as it sounds, since it may require truncating the longer one in the middle of a note (or notes), and it may be that one (or both) of the *Music* values is infinite.

The goal is to define a "truncating parallel composition" operator $(/{=}{:}) :: \mathit{Music}\ a \rightarrow \mathit{Music}\ a \rightarrow \mathit{Music}\ a$. Using *takeM*, we can make an initial attempt at a suitable definition for $(/{=}{:})$ as follows:

$$
\begin{array}{ll}
(/{=}{:}) & :: \mathit{Music}\ a \rightarrow \mathit{Music}\ a \rightarrow \mathit{Music}\ a\\
\multicolumn{2}{l}{m_1\ /{=}{:}\ m_2 = \mathit{takeM}\ (\mathit{dur}\ m_2)\ m_1 :=: \mathit{takeM}\ (\mathit{dur}\ m_1)\ m_2}
\end{array}
$$

Unfortunately, whereas *takeM* can handle infinite-duration music values, $(/{=}{:})$ cannot. This is because $(/{=}{:})$ computes the duration of both of its arguments, but if one of them, say m_1, has infinite duration, then $\mathit{dur}\ m_1 \Rightarrow \bot$. If, in a particular context, we know that only one of the two arguments is infinite, and we know which one (say m_1), it is always possible to write:

$$\mathit{takeM}\ (\mathit{dur}\ m_2)\ m_1 :=: m_2$$

But somehow this seems unsatisfactory.

6.9.1 Lazy Evaluation to the Rescue

The root of this problem is that *dur* uses a conventional number type, namely the type *Rational* (which is a ratio of *Integer*s), to compute with, which does not have a value for infinity ($\perp$ is not the same as infinity!). But what if we were to somehow compute the duration *lazily*—meaning that we only compute that much of the duration that is needed to perform some arithmetic result of interest. In particular, if we have one number n that we know is "at least" x, and another number m that is exactly y, then if $x > y$, we know that $n > m$, even if n's actual value is infinity!

To realize this idea, let's first define a type synonym for "lazy durations:"

$$\textbf{type}\ \mathit{LazyDur} = [\mathit{Dur}]$$

The intent is that a value $d :: \mathit{LazyDur}$ is a non-decreasing list of durations such that the last element in the list is the actual duration, and an infinite list implies an infinite duration.

Now let's define a new verion of *dur* that computes the *LazyDur* of its argument:

```
durL :: Music a → LazyDur
durL m@(Prim _)            = [dur m]
durL (m1 :+: m2)           = let d1 = durL m1
                             in d1 ++ map (+(last d1)) (durL m2)
durL (m1 :=: m2)           = mergeLD (durL m1) (durL m2)
durL (Modify (Tempo r) m) = map (/r) (durL m)
durL (Modify _ m)          = durL m
```

where *mergeLD* merges two *LazyDur* values into one:

```
mergeLD :: LazyDur → LazyDur → LazyDur
mergeLD [] ld = ld
mergeLD ld [] = ld
mergeLD ld1@(d1 : ds1) ld2@(d2 : ds2) =
  if d1 < d2 then d1 : mergeLD ds1 ld2
             else  d2 : mergeLD ld1 ds2
```

We can then define a function *minL* to compare a *LazyDur* with a regular *Dur*, returning the least *Dur* as a result:

```
minL :: LazyDur → Dur → Dur
minL []       d' = d'
minL [d]      d' = min d d'
minL (d : ds) d' = if d < d' then minL ds d' else d'
```

And with *minL* we can then define a new version of *takeM*:

$$
\begin{array}{ll}
\mathit{takeML} :: \mathit{LazyDur} \rightarrow \mathit{Music}\ a \rightarrow \mathit{Music}\ a & \\
\mathit{takeML}\ [\,]\ m & = \mathit{rest}\ 0 \\
\mathit{takeML}\ (d : ds)\ m \mid d \leqslant 0 & = \mathit{takeML}\ ds\ m \\
\mathit{takeML}\ ld\ (\mathit{Prim}\ (\mathit{Note}\ oldD\ p)) & = \mathit{note}\ (\mathit{minL}\ ld\ oldD)\ p \\
\mathit{takeML}\ ld\ (\mathit{Prim}\ (\mathit{Rest}\ oldD)) & = \mathit{rest}\ (\mathit{minL}\ ld\ oldD) \\
\mathit{takeML}\ ld\ (m_1 :=: m_2) & = \mathit{takeML}\ ld\ m_1 :=: \mathit{takeML}\ ld\ m_2 \\
\mathit{takeML}\ ld\ (m_1 :+: m_2) & = \\
\quad \mathbf{let}\ m'_1 = \mathit{takeML}\ ld\ m_1 & \\
\qquad\quad m'_2 = \mathit{takeML}\ (\mathit{map}\ (\lambda d \rightarrow d - \mathit{dur}\ m'_1)\ ld)\ m_2 & \\
\quad \mathbf{in}\ m'_1 :+: m'_2 & \\
\mathit{takeML}\ ld\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ m) & = \mathit{tempo}\ r\ (\mathit{takeML}\ (\mathit{map}\ (*r)\ ld)\ m) \\
\mathit{takeML}\ ld\ (\mathit{Modify}\ c\ m) & = \mathit{Modify}\ c\ (\mathit{takeML}\ ld\ m)
\end{array}
$$

Compare this definition with that of *takeM*—they are very similar.

Finally, we can define a correct (meaning it works properly on infinite *Music* values) version of (/=:) as follows:

$$
\begin{array}{l}
(/=:) \qquad :: \mathit{Music}\ a \rightarrow \mathit{Music}\ a \rightarrow \mathit{Music}\ a \\
m_1\ /=:\ m_2 = \mathit{takeML}\ (\mathit{durL}\ m_2)\ m_1 :=: \mathit{takeML}\ (\mathit{durL}\ m_1)\ m_2
\end{array}
$$

Whew! This may seem like a lot of effort, but the new code is actually not much different from the old, and now we can freely use (/=:) without worrying about which if any of its arguments are infinite.

Exercise 6.5 Try using (/=:) with some infinite *Music* values (such as created by *repeatM*) to assure yourself that it works properly. When using it with *two* infinite values, it should return an infinite value, which you can test by applying *takeM* to the result.

6.10 Trills

A *trill* is an ornament that alternates rapidly between two (usually adjacent) pitches. Two versions of a trill function will be defined, 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 (Prim (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."
```

Using this function it is simple to define a version 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
```

Figure 6.3 shows a nice use of the trill functions in encoding the opening lines of John Philip Sousa's *Stars and Stripes Forever*.

Details: *ssfMel* uses a **where** clause, which is similar to a **let** expression, except that the equations appear after the result, rather than before.

$$
\begin{array}{l}
\mathit{ssfMel} :: \mathit{Music\ Pitch} \\
\mathit{ssfMel} = \mathit{line}\ (l_1 +\!\!+\ l_2 +\!\!+\ l_3 +\!\!+\ l_4) \\
\quad \textbf{where}\ l_1 = [\mathit{trilln}\ 2\ 5\ (\mathit{bf}\ 6\ \mathit{en}), \mathit{ef}\ 7\ \mathit{en}, \mathit{ef}\ 6\ \mathit{en}, \mathit{ef}\ 7\ \mathit{en}] \\
\qquad\qquad l_2 = [\mathit{bf}\ 6\ \mathit{sn}, c\ 7\ \mathit{sn}, \mathit{bf}\ 6\ \mathit{sn}, g\ 6\ \mathit{sn}, \mathit{ef}\ 6\ \mathit{en}, \mathit{bf}\ 5\ \mathit{en}] \\
\qquad\qquad l_3 = [\mathit{ef}\ 6\ \mathit{sn}, f\ 6\ \mathit{sn}, g\ 6\ \mathit{sn}, \mathit{af}\ 6\ \mathit{sn}, \mathit{bf}\ 6\ \mathit{en}, \mathit{ef}\ 7\ \mathit{en}] \\
\qquad\qquad l_4 = [\mathit{trill}\ 2\ \mathit{tn}\ (\mathit{bf}\ 6\ \mathit{qn}), \mathit{bf}\ 6\ \mathit{sn}, \mathit{denr}] \\
\mathit{starsAndStripes} :: \mathit{Music\ Pitch} \\
\mathit{starsAndStripes} = \mathit{instrument\ Flute\ ssfMel}
\end{array}
$$

Figure 6.3: Trills in *Stars and Stripes Forever*

6.11 Grace Notes

Recall from Chapter 4 the function *graceNote* to generate grace notes. A more general version is defined below, which takes a *Rational* argument that specifies that fraction of the principal note's duration to be used for the grace note's duration:

$$
\begin{array}{l}
\mathit{grace} :: \mathit{Int} \rightarrow \mathit{Rational} \rightarrow \mathit{Music\ Pitch} \rightarrow \mathit{Music\ Pitch} \\
\mathit{grace}\ n\ r\ (\mathit{Prim}\ (\mathit{Note}\ d\ p)) = \\
\quad \mathit{note}\ (r * d)\ (\mathit{trans}\ n\ p) :\!+\!: \mathit{note}\ ((1 - r) * d)\ p \\
\mathit{grace}\ n\ r\ \_ \qquad\qquad\qquad = \\
\quad \mathit{error}\ \texttt{"grace: can only add a grace note to a note"}
\end{array}
$$

Thus *grace* $n\ r\ (note\ d\ p)$ is a *Music* value consisting of two notes, the first being the grace note whose duration is $r * d$ and whose pitch is n semitones higher (or lower if n is negative) than p, and the second being the principal note at pitch p but now with duration $(1 - r) * d$.

Note that *grace* places the downbeat of the grace note at the point written for the principal note. Sometimes the interpretation of a grace note is such that the downbeat of the principal note is to be unchanged. In that case, the grace note reduces the duration of the *previous* note. We can define a function *grace2* that takes two notes as arguments, and places the grace note appropriately:

$$
\begin{array}{l}
\mathit{grace2} :: \mathit{Int} \rightarrow \mathit{Rational} \rightarrow \\
\qquad\qquad \mathit{Music\ Pitch} \rightarrow \mathit{Music\ Pitch} \rightarrow \mathit{Music\ Pitch}
\end{array}
$$

$$grace2\ n\ r\ (Prim\ (Note\ d_1\ p_1))\ (Prim\ (Note\ d_2\ p_2)) =$$
$$\quad note\ (d_1 - r * d_2)\ p_1 :+: note\ (r * d_2)\ (trans\ n\ p_2) :+: note\ d_2\ p_2$$
$$grace2\ \_\ \_\ \_\ \_ =$$
$$\quad error\ \texttt{"grace2: can only add a grace note to a note"}$$

Exercise 6.6 Related to trills and grace notes in Western classical music are the notions of *mordent*, *turn*, and *appoggiatura*. Define functions to realize these musical ornamentations.

6.12 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, although 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, it is worth defining some handy functions for using the General MIDI Standard. In Figure 6.4 a data type is defined that 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 Euterpea terminology. So all that remains to be done is a way to convert these percussion sound names into a *Music* value; i.e. a *Note*:

$$perc :: PercussionSound \rightarrow Dur \rightarrow Music\ Pitch$$
$$perc\ ps\ dur = note\ dur\ (pitch\ (fromEnum\ ps + 35))$$

```
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
```

Figure 6.4: General MIDI Percussion Names

> **Details:** *fromEnum* is an operator in the *Enum* class, which is all about enumerations, and will be discussed in more detail in Chapter 7. 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.

If a *Music* value returned from *perc* is played using a piano sound, then you will get a piano sound. But if you specify the instrument *Percussion*, MIDI knows to play the apppropriate *PercussionSound*.

Recall the *InstrumentName* data type from Chapter 2. If a *Music* value returned from *perc* is played using, say, the *AcousticGrandPiano* instrument, then you will hear an acounstic grand piano sound at the appropriate pitch. But if you specify the *Percussion* instrument, then you will hear the percussion sound that was specified as an argument to *perc*.

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

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

Exercise 6.7 Write a program that generates all of the General MIDI percussion sounds, playing through each of them one at a time.

Exercise 6.8 Find a drum beat that you like, and express it in Euterpea. Then use *repeatM*, *takeM*, and (:=:) to add a simple melody to it.

6.13 A Map for Music

Recall from Chapter 3 the definition of *map*:

$$
\begin{array}{ll}
map & :: (a \to b) \to [a] \to [b] \\
map\ f\ [\,] & = [\,] \\
map\ f\ (x : xs) & = f\ x : map\ f\ xs
\end{array}
$$

This function is defined on the list data type. Is there something analogous for *Music*? I.e. a function:[1]

$$mMap :: (a \to b) \to Music\ a \to Music\ b$$

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

$$
\begin{array}{ll}
pMap & :: (a \to b) \to Primitive\ a \to Primitive\ b \\
pMap\ f\ (Note\ d\ x) & = Note\ d\ (f\ x) \\
pMap\ f\ (Rest\ d) & = Rest\ d
\end{array}
$$

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

$$
\begin{array}{ll}
mMap & :: (a \to b) \to Music\ a \to Music\ b \\
mMap\ f\ (Prim\ p) & = Prim\ (pMap\ f\ p) \\
mMap\ f\ (m_1 :+: m_2) & = mMap\ f\ m_1 :+: mMap\ f\ m_2 \\
mMap\ f\ (m_1 :=: m_2) & = mMap\ f\ m_1 :=: mMap\ f\ m_2 \\
mMap\ f\ (Modify\ c\ m) & = Modify\ c\ (mMap\ f\ m)
\end{array}
$$

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, let's introduces a *Volume* type for a note:

$$\textbf{type}\ Volume = Int$$

The goal is to convert a value of type *Music Pitch* into a value of type *Music* (*Pitch*, *Volume*)—that is, to pair each pitch with a volume attribute. We can define a function to do so as follows:

$$
\begin{array}{l}
addVolume\ \ :: Volume \to Music\ Pitch \to Music\ (Pitch, Volume) \\
addVolume\ v = mMap\ (\lambda p \to (p, v))
\end{array}
$$

For MIDI, the variable v can range from 0 (softest) to 127 (loudest).

For example, compare the loudness of these two phrases:

$$
\begin{array}{l}
m_1, m_2 :: Music\ (Pitch, Volume) \\
m_1 = addVolume\ 100\ (c\ 4\ qn :+: d\ 4\ qn :+: e\ 4\ qn :+: c\ 4\ qn)
\end{array}
$$

[1] The name *mapM* would perhaps have been a better choice here, to be consistent with previous names. However, *mapM* is a predefined function in Haskell, and thus *mMap* is used instead. Similarly, Haskell's *Monad* library defines a function *foldM*, and thus in the next section the name *mFold* is used instead.

$$m_2 = addVolume\ 30\ (c\ 4\ qn :+: d\ 4\ qn :+: e\ 4\ qn :+: c\ 4\ qn)$$

using the *play* function. (Recall from Section 2.3 that the type of the argument to *play* must be made clear, as is done here with the type signature.)

> **Details:** Note that the name *Volume* is used both as a type synonym and as a constructor—Haskell allows this, since they can always be distinguished by context.

Exercise 6.9 Using *mMap*, define a function:

$$\begin{aligned} scaleVolume :: Rational &\rightarrow Music\ (Pitch, Volume) \\ &\rightarrow Music\ (Pitch, Volume) \end{aligned}$$

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

(This problem requires multiplying a *Rational* number by an *Int* (i.e. *Volume*). To do this, some coercions between number types are needed, which in Haskell is done using *qualified types*, which are discussed in Chapter 7. For now, you can simply do the following: If *v* is the volume of a note, then *round* $(s * fromIntegral\ v)$ is the desired scaled volume.)

6.14 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 (*Prim*, (:+:), (:=:), and *Modify*). Thus the following function takes four arguments in addition to the *Music* value it is transforming, instead of two:

$$
\begin{aligned}
&\mathit{mFold} :: (\mathit{Primitive}\ a \to b) \to (b \to b \to b) \to (b \to b \to b) \to \\
&\qquad\qquad (\mathit{Control} \to b \to b) \to \mathit{Music}\ a \to b \\
&\mathit{mFold}\ f\ (+:)\ (=:)\ g\ m = \\
&\quad \mathbf{let}\ \mathbf{rec} = \mathit{mFold}\ f\ (+:)\ (=:)\ g \\
&\quad \mathbf{in\ case}\ m\ \mathbf{of} \\
&\qquad \mathit{Prim}\ p \quad\;\;\; \to f\ p \\
&\qquad m_1 :+: m_2 \;\; \to \mathbf{rec}\ m_1 +: \mathbf{rec}\ m_2 \\
&\qquad m_1 :=: m_2 \;\; \to \mathbf{rec}\ m_1 =: \mathbf{rec}\ m_2 \\
&\qquad \mathit{Modify}\ c\ m \to g\ c\ (\mathbf{rec}\ m)
\end{aligned}
$$

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

$$\mathit{mFold}\ \mathit{Prim}\ (:+:)\ (:=:)\ \mathit{Modify}\ m == m$$

Although intuitive, proving this property requires induction, a proof technique discussed in Chapter 10.

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:

$$
\begin{aligned}
&\mathit{mMap} \quad :: (a \to b) \to \mathit{Music}\ a \to \mathit{Music}\ b \\
&\mathit{mMap}\ f = \mathit{mFold}\ g\ (:+:)\ (:=:)\ \mathit{Modify}\ \mathbf{where} \\
&\quad g\ (\mathit{Note}\ d\ x) = \mathit{note}\ d\ (f\ x) \\
&\quad g\ (\mathit{Rest}\ d) \quad\; = \mathit{rest}\ d
\end{aligned}
$$

More interestingly, we can use *mFold* to more succinctly define functions such as *dur* from Section 6.5:

$$
\begin{aligned}
&\mathit{dur} :: \mathit{Music}\ a \to \mathit{Dur} \\
&\mathit{dur} = \mathit{mFold}\ \mathit{getDur}\ (+)\ \mathit{max}\ \mathit{modDur}\ \mathbf{where} \\
&\quad \mathit{getDur}\ (\mathit{Note}\ d\ \_) \quad\; = d \\
&\quad \mathit{getDur}\ (\mathit{Rest}\ d) \quad\;\;\; = d \\
&\quad \mathit{modDur}\ (\mathit{Tempo}\ r)\ d = d / r \\
&\quad \mathit{modDur}\ \_\ d \qquad\qquad = d
\end{aligned}
$$

Exercise 6.10 Redefine *revM* from Section 6.6 using *mFold*.

Exercise 6.11 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 is "musically equivalent" to (i.e. sounds the same as) $insideOut\ m$ and (b) find a value $m :: Music\ Pitch$ such that $m :+: insideOut\ m :+: m$ sounds interesting. (You are free to define what "sounds interesting" means.)

6.15 Crazy Recursion

With all the functions and data types that have been defined, and the power of recursion and higher-order functions well understood, we can start to do some wild and crazy things with music. Here is just one such idea.

The goal is to define a function to recursively apply transformations f (to elements in a sequence) and g (to accumulated phrases) some specified number of times:

$$\begin{aligned}
&rep :: (Music\ a \rightarrow Music\ a) \rightarrow (Music\ a \rightarrow Music\ a) \rightarrow Int \\
&\qquad \rightarrow Music\ a \rightarrow Music\ a \\
&rep\ f\ g\ 0\ m = rest\ 0 \\
&rep\ f\ g\ n\ m = m :=: g\ (rep\ f\ g\ (n-1)\ (f\ m))
\end{aligned}$$

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

$$\begin{aligned}
run\quad &= rep\ (transpose\ 5)\ (delayM\ tn)\ 8\ (c\ 4\ tn) \\
cascade\ &= rep\ (transpose\ 4)\ (delayM\ en)\ 8\ run \\
cascades &= rep\ id\ (delayM\ sn)\ 2\ cascade
\end{aligned}$$

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

$$final = cascades :+: revM\ cascades$$

What happens if the f and g arguments are reversed?

$$\begin{aligned}
run' &= rep\ (delayM\ tn)\ (transpose\ 5)\ 8\ (c\ 4\ tn) \\
cascade' &= rep\ (delayM\ en)\ (transpose\ 4)\ 8\ run' \\
cascades' &= rep\ (delayM\ sn)\ id\ 2\ cascade' \\
final' &= cascades' :+: revM\ cascades'
\end{aligned}$$

Exercise 6.12 Consider this sequence of 8 numbers:

$s_1 = [1, 5, 3, 6, 5, 0, 1, 1]$

We might interpret this as a sequence of pitches, i.e. a melody. Another way to represent this sequence is as a sequence of 7 intervals:

$s_2 = [4, -2, 3, -1, -5, 1, 0]$

Together with the starting pitch (i.e. 1), this sequence of intervals can be used to reconstruct the original melody. But, with a suitable transposition to eliminate negative numbers, it can also be viewed as another melody. Indeed, we can repeat the process: s_2 can be represented by this sequence of 6 intervals:

$s_3 = [-6, 5, -4, -4, 6, -1]$

Together with the starting number (i.e. 4), s_3 can be used to reconstruct s_2. Continuing the process:

$$
\begin{aligned}
s_4 &= [11, -9, 0, 10, -7]\\
s_5 &= [-20, 9, 10, -17]\\
s_6 &= [29, 1, -27]\\
s_7 &= [-28, -28]\\
s_8 &= [0]
\end{aligned}
$$

Now, if we take the first element of each of these sequences to form this 8-number sequence:

$ic = [0, -28, 29, -20, 11, -6, 4, 1]$

then it alone can be used to re-create the original 8-number sequence in its entirety. Of course, it can also be used as the original melody was used, and we could derive another 8-note sequence from it—and so on. The list ic will be referred to as the "interval closure" of the original list s_1.

Your job is to:

a) Define a function *toIntervals* that takes a list of n numbers, and generates a list of n lists, such that the i^{th} list is the sequence s_i as defined above.

b) Define a function *getHeads* that takes a list of n lists and returns a list of n numbers such that the i^{th} element is the head of the i^{th} list.

c) Compose the above two functions in a suitable way to define a function *intervalClosure* that takes an n-element list and returns its interval closure.

d) Define a function *intervalClosures* that takes an n-element list and returns an infinite sequence of interval closures.

e) Now for the open-ended part of this exercise: Interpret the outputs of any of the functions above to create some "interesting" music.

Exercise 6.13 Write a Euterpea program that sounds like an infinitely descending (in pitch) sequence of musical lines. Each descending line should fade into the audible range as it begins its descent, and then fade out as it descends further. So the beginning and end of each line will be difficult to hear. And there will be many such lines, each starting at a different time, some perhaps descending a little faster than others, or perhaps using different instrument sounds, and so on. The effect will be that as the music is listened to, everything will seem to be falling, falling, falling with no end, but no beginning either. (This illusion is called the *Shepard Tone*, or *Shepard Scale*, first introduced by Roger Shepard in 1964 [She64].)

Use high-order functions, recursion, and whatever other abstraction techniques you have learned to write an elegant solution to this problem. Try to parameterize things in such a way that, for example, with a simple change, you could generate an infinite *ascension* as well. The *Volume* constructor in the *NoteAttribute* type, as used in the definition of *addVol*, should be used to set the volumes.

Exercise 6.14 Do something wild and crazy with Euterpea.

Chapter 7

Qualified Types and Type Classes

This chapter introduces the notions of *qualified types* and *type classes*. These concepts can be viewed as a refinement of the notion of polymorphism, and increase the ability to write modular programs.

7.1 Motivation

A polymorphic type such as $(a \to a)$ can be viewed as shorthand for $\forall(a) a \to a$, 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 is probably not a good idea to limit $(+)$ to a *single* (that is, *monomorphic*) type such as $Integer \to Integer \to Integer$, since there are other kinds of numbers—such as rational and floating-point numbers—that we would like to perform addition on as well. Nor is it a good idea to have a different addition function for each number type, since that would require giving each a different name, such as *addInteger*, *addRational*, *addFloat*, etc. And, unfortunately, giving $(+)$ a type such as $a \to a \to a$ will not work, since this would imply that we could add things other than numbers, such as characters, pitch classes, lists, tuples, functions, and any type that we might define on our own!

Haskell provides a solution to this problem through the use of *qualified* *types*. Conceptually, it is helpful to think of a qualified type just as a polymorphic type, except that in place of "*for all* types a" it will be possible to say "for all types *a that are members of the type class C*," where the type class C can be thought of as a set of types. For example, suppose there is a type class Num with members $Integer$, $Rational$, and $Float$. Then an accurate type for $(+)$ would be $\forall(a{\in}Num)a \rightarrow a \rightarrow a$. But in Haskell, instead of writing $\forall(a \in Num)\cdots$, the notation $Num\ a \Rightarrow\cdots$ is used. So the proper type signature for $(+)$ is:

$$(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

which should be read: "for all types a that are members of the type class Num, $(+)$ has type $a \rightarrow a \rightarrow a$." Members of a type class are also called *instances* of the class, and these two terms will be used 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.

Recall now the type signature given for the function *simple* in Chapter 1:

$$\begin{aligned} &simple &&:: Integer \rightarrow Integer \rightarrow Integer \rightarrow Integer \\ &simple\ x\ y\ z &&= x * (y + z) \end{aligned}$$

Note that *simple* uses the operator $(+)$ discussed above. It also uses $(*)$, whose type is the same as that for $(+)$:

$$(*) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$$

This suggests that a more general type for *simple* is:

$$\begin{aligned} &simple &&:: Num\ a \Rightarrow a \rightarrow a \rightarrow a \rightarrow a \\ &simple\ x\ y\ z &&= x * (y + z) \end{aligned}$$

Indeed, this is the preferred, most general type that can be given for *simple*. It can now be used with any type that is a member of the Num class, which includes $Integer$, Int, $Rational$, $Float$ and $Double$, among others.

The ability to qualify polymorphic types is a unique feature of Haskell, and, as we will soon see, provides great expressiveness. In the following sections the idea is explored much more thoroughly, and in particular it is shown how a programmer can define his or her own type classes and their instances. To begin, let's take a closer look at one of the pre-defined type classes in Haskell, having to do with equality.

7.2 Equality

Equality between two expressions e_1 and e_2 in Haskell means that the value of e_1 is the same as the value of e_2. Another way to view equality is that we should be able to substitute e_1 for e_2, or vice versa, 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, two infinite *Music* values, or two functions of type $Integer \rightarrow Integer$.[1] 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 it might not make sense. For example, it is common to compare two characters, two integers, two floating-point numbers, etc. On the other hand, comparing the equality of infinite data structures, or 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:

$$(==) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

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:

$$\begin{array}{lll}
42 & == 42 & \Rightarrow True \\
42 & == 43 & \Rightarrow False \\
\texttt{'a'} & == \texttt{'a'} & \Rightarrow True \\
\texttt{'a'} & == \texttt{'b'} & \Rightarrow False
\end{array}$$

Furthermore, the expression $42 == \texttt{'a'}$ is *ill-typed*; Haskell is clever enough

[1]This is the same as determining *program equivalence*, a well-known example of an *undecideable problem* in the theory of computation.

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 $(Integer, Char)$, $[Integer]$, $[Float]$, etc. Thus:

$$\begin{array}{lll} [42,43] & == [42,43] & \Rightarrow True \\ [4.2,4.3] & == [4.3,4.2] & \Rightarrow False \\ (42,\texttt{'a'}) & == (42,\texttt{'a'}) & \Rightarrow True \end{array}$$

This will be elaborated upon in a later section.

Type constraints also propagate through function definitions. For example, consider this definition of the function $\in$ that tests for membership in a list:

$$\begin{array}{ll} x \in [\,] & = False \\ x \in (y : ys) & = x == y \vee x \in ys \end{array}$$

Details: $(\in)$ is actually written *elem* in Haskell; i.e. it is a normal function, not an infix operator. Of course it can be used in an infix manner (and it often is) by enclosing it in backquotes.

Note the use of $(==)$ on the right-hand side of the second equation. The principal type for $(\in)$ is thus:

$$\in ::Eq\ a \Rightarrow a \rightarrow [\,a\,] \rightarrow Bool$$

This should be read, "For every type a that is an instance of the class *Eq*, $(\in)$ has type $a \rightarrow [\,a\,] \rightarrow Bool$." This is exactly what we would hope for—it expresses the fact that $(\in)$ is not defined on all types, just those for which computational equality is defined.

The above type for $(\in)$ is also its principal type, and Haskell will infer this type if no signature is given. Indeed, if we were to write the type signature:

$$(\in) :: a \rightarrow [\,a\,] \rightarrow Bool$$

a type error would result, because this type is fundamentally *too general*, and the Haskell type system will complain.

Details: On the other hand, we could write:

$$(\in) :: Integer \rightarrow [Integer] \rightarrow Bool$$

if we expect to use $(\in)$ 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:

$$square\ 42\ \Rightarrow 1764$$
$$square\ 4.2 \Rightarrow 17.64$$

The *Num* class will be discusssed in greater detail shortly.

7.3 Defining Our Own Type Classes

Haskell provides a mechanism whereby we can create our 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, consider the *Eq* class. It is created by the following *type class declaration*:

```
class Eq a where
  (==) :: a -> a -> Bool
```

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." $(==)$ is called an *operation* in the class *Eq*, and in general more than one operation is allowed in a class. More examples of this will be introduced 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 we have defined on our 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 | Ff | Es | 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 thirty equations needed to make this definition complete. Indeed, this is rather tedious! It is not only tedious, it is also dead obvious how (==) should be defined.

7.3.1 Derived Instances

To alleviate the burden of defining instances such as above, Haskell provides a convenient way to *automatically derive* such instance declarations from data type declarations, for certain predefined type classes. This is done using a **deriving** clause. For example, in the case of *PitchClass* we can simply write:

$$\begin{aligned}&\textbf{data}\ \mathit{PitchClass} = \mathit{Cff} \mid \mathit{Cf} \mid C \mid \mathit{Dff} \mid \mathit{Cs} \mid \mathit{Df} \mid \mathit{Css} \mid D \mid \mathit{Eff} \mid \mathit{Ds}\\ &\qquad\qquad\qquad \mid \mathit{Ef} \mid \mathit{Fff} \mid \mathit{Dss} \mid E \mid \mathit{Ff} \mid \mathit{Es} \mid F \mid \mathit{Gff} \mid \mathit{Ess} \mid \mathit{Fs}\\ &\qquad\qquad\qquad \mid \mathit{Gf} \mid \mathit{Fss} \mid G \mid \mathit{Aff} \mid \mathit{Gs} \mid \mathit{Af} \mid \mathit{Gss} \mid A \mid \mathit{Bff} \mid \mathit{As}\\ &\qquad\qquad\qquad \mid \mathit{Bf} \mid \mathit{Ass} \mid B \mid \mathit{Bs} \mid \mathit{Bss}\\ &\quad \textbf{deriving}\ \mathit{Eq}\end{aligned}$$

With this declaration, Haskell will automatically derive the instance declaration given above, so that $(==)$ behaves in the way we would expect it to.

Consider now a polymorphic type, such as the *Primitive* type from Chapter 2:

$$\begin{aligned}&\textbf{data}\ \mathit{Primitive}\ a = \mathit{Note}\ \mathit{Dur}\ a\\ &\qquad\qquad\qquad\quad \mid \mathit{Rest}\ \mathit{Dur}\end{aligned}$$

What should an instance for this type in the class *Eq* look like? Here is a first attempt:

$$\begin{aligned}&\textbf{instance}\ \mathit{Eq}\ (\mathit{Primitive}\ a)\ \textbf{where}\\ &\quad \mathit{Note}\ d_1\ x_1 == \mathit{Note}\ d_2\ x_2 = (d_1 == d_2) \land (x_1 == x_2)\\ &\quad \mathit{Rest}\ d_1 \quad\ == \mathit{Rest}\ d_2 \quad\ = d_1 == d_2\\ &\quad \_ \qquad\qquad == \_ \qquad\qquad = \mathit{False}\end{aligned}$$

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 predefined 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 $x_1 == x_2$? x_1 and x_2 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 this is not known in general. The simple fix is to add a constraint to the instance declaration, as follows:

$$\begin{aligned}&\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Primitive}\ a)\ \textbf{where}\\ &\quad \mathit{Note}\ d_1\ x_1 == \mathit{Note}\ d_2\ x_2 = (d_1 == d_2) \land (x_1 == x_2)\\ &\quad \mathit{Rest}\ d_1 \quad\ == \mathit{Rest}\ d_2 \quad\ = d_1 == d_2\\ &\quad \_ \qquad\qquad == \_ \qquad\qquad = \mathit{False}\end{aligned}$$

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, if we had written the original type declaration like this:

$$\begin{aligned}&\textbf{data}\ \mathit{Primitive}\ a = \mathit{Note}\ \mathit{Dur}\ a\\ &\qquad\qquad\qquad\quad \mid \mathit{Rest}\ \mathit{Dur}\\ &\quad \textbf{deriving}\ \mathit{Eq}\end{aligned}$$

then Haskell would have derived the above correct instance declaration 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 the effort above, (b) *Octave* is a synonym for *Int*, which is a predefined instance of *Eq*, and (c) as mentioned earlier the pair type is a predefined instance of *Eq*. Indeed, now that an instance for a polymorphic type has been seen, we can understand what the predefined instance for polymorphic pairs must look like, namely:

$$
\begin{array}{l}
\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a,b)\ \textbf{where} \\
\quad (x_1, y_1) == (x_2, y_2) = (x_1 == x_2) \wedge (y_1 == y_2)
\end{array}
$$

About the only thing not considered is a *recursive* data type. For example, recall the *Music* data type, also from Chapter 2:

$$
\begin{array}{lll}
\textbf{data}\ Music\ a & = & Prim\ (Primitive\ a) \\
 & \mid & Music\ a :\!+\!: Music\ a \\
 & \mid & Music\ a :\!=\!: Music\ a \\
 & \mid & Modify\ Control\ (Music\ a)
\end{array}
$$

Its instance declaration for *Eq* seems obvious:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{instance}\ Eq\ a \Rightarrow Eq\ (Music\ a)\ \textbf{where}} \\
\quad Prim\ p_1 & == Prim\ p_2 & = p_1 == p_2 & \\
\quad (m_1 :\!+\!: m_2) & == (m_3 :\!+\!: m_4) & = (m_1 == m_3) & \wedge\ (m_2 == m_4) \\
\quad (m_1 :\!=\!: m_2) & == (m_3 :\!=\!: m_4) & = (m_1 == m_3) & \wedge\ (m_2 == m_4) \\
\quad Modify\ c_1\ m_1 & == Modify\ c_2\ m_2 & = (c_1 == c_2) & \wedge\ (m_1 == m_2)
\end{array}
$$

Indeed, assuming that *Control* is an instance of *Eq*, this is just what is expected, and can be automatically derived by adding a **deriving** clause to the data type declaration for *Music*.

7.3.2 Default Methods

In reality, the class *Eq* as defined in Haskell's Standard Prelude is slightly richer than what is defined above. Here it is in its exact form:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{class}\ Eq\ a\ \textbf{where}} \\
\multicolumn{2}{l}{\quad (==), (\neq) :: a \to a \to Bool} \\
\quad x \neq y & = \neg\ (x == y) \\
\quad x == y & = \neg\ (x \neq y)
\end{array}
$$

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 would expect: 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 instead will have to use the operator `/=` and the function name `not`, respectively.

A useful slogan that helps to distinguish type classes from ordinary polymorphism is this: "polymorphism captures similar structure over different values, while type classes capture similar operations over different structures." For example, a sequences of integers, sequence of characters, etc. can be captured as a polymorphic *List*, whereas equality of integers, equality of trees, etc. can be captured by a type class such as *Eq*.

7.3.3 Inheritance

Haskell also supports a notion called *inheritance*. For example, we may wish to define a class *Ord* that "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 Appendix B):

$$\begin{array}{l}
\mathbf{class}\ \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a\ \mathbf{where} \\
\quad (<), (\leqslant), (\geqslant), (>) :: a \to a \to \mathit{Bool} \\
\quad \mathit{max}, \mathit{min} \qquad\qquad\ :: a \to a \to a
\end{array}$$

Note the constraint $\mathit{Eq}\ a \Rightarrow$ in the **class** declaration. *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 used for *Eq*, we could declare *Music* an instance of *Ord* as follows (note the constraint $Ord\ a \Rightarrow ...$):

$$
\begin{array}{l}
\textbf{instance}\ Ord\ a \Rightarrow Ord\ (Music\ a)\ \textbf{where} \\
\quad \begin{array}{llll}
Prim\ p_1 & < Prim\ p_2 & = p_1 < p_2 & \\
(m_1 :+: m_2) & < (m_3 :+: m_4) & = (m_1 < m_3) & \wedge (m_2 < m_4) \\
(m_1 :=: m_2) & < (m_3 :=: m_4) & = (m_1 < m_3) & \wedge (m_2 < m_4) \\
Modify\ c_1\ m_1 & < Modify\ c_2\ m_2 & = (c_1 < c_2) & \wedge (m_1 < m_2) \\
...
\end{array}
\end{array}
$$

Although this is a perfectly well-defined definition for $<$, it is not clear that it exhibits the desired behavior, an issue that will be returned to in Section 7.7.

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 just 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 that 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 overall constraint in the class declaration.

Type Class	**Key functions**	**Key instances**
Num	$(+), (-), (*) :: Num\ a \Rightarrow a \to a \to a$ $negate :: Num\ a \Rightarrow a \to a$ minimal set: all but $(-)$ or *negate*	*Integer*, *Int*, *Float*, *Double*, *Rational*
Eq	$(==), (\neq) :: Eq\ a \Rightarrow a \to a \to Bool$ minimal set: either $(==)$ or $(\neq)$	*Integer*, *Int*, *Float*, *Double*, *Rational*, *Char*, *Bool*, ...
Ord	$(>), (<), (\geqslant), (\leqslant) ::$ $Ord\ a \Rightarrow a \to a \to Bool$ $max, min :: Ord\ a \Rightarrow a \to a \to a$ minimal set: $(\leqslant)$	*Integer*, *Int*, *Float*, *Double*, *Rational*, *Char*, *Bool*, ...
Enum	$succ, pred :: Enum\ a \Rightarrow a \to a$ $fromEnum :: Enum\ a \Rightarrow a \to Int$ $toEnum :: Enum\ a \Rightarrow Int \to a$ also enables arithmetic sequences minimal set: *toEnum* & *fromEnum*	*Integer*, *Int*, *Float*, *Double*, *Rational*, *Char*, *Bool*, ...
Bounded	$minBound, maxBound :: a$	*Int*, *Char*, *Bool*
Show	$show :: Show\ a \Rightarrow a \to String$	Almost every type except for functions
Read	$read :: Read\ a \Rightarrow String \to a$	Almost every type except for functions

Figure 7.1: Common Type Classes and Their Instances

7.4 Haskell's Standard Type Classes

The Standard Prelude defines many useful type classes, including *Eq* and *Ord*. They are described in detail in Appendix B. In addition, the Haskell Report and the Library Report contain useful examples and discussions of type classes; you are encouraged to read through them.

Most of the standard type classes in Haskell are shown in Figure 7.1, along with their key instances. Since each of these has various default mth-ods defined, also shown is the minimal set of methods that must defined—the rest are taken care of by the default methods. For example, for *Ord*, all we have to provide is a definition for $(\leqslant)$.

The *Num* class, which has been used 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.

7.4.1 The *Num* Class

As we already know, Haskell provides several kinds of numbers, some of which have already been introduced: *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 are not 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:

$$
\begin{array}{l}
\textbf{class}\ (Eq\ a, Show\ a) \Rightarrow Num\ a\ \textbf{where} \\
\quad \begin{array}{ll}
(+), (-), (*) & :: a \rightarrow a \rightarrow a \\
negate & :: a \rightarrow a \\
abs, signum & :: a \rightarrow a \\
fromInteger & :: Integer \rightarrow a
\end{array}
\end{array}
$$

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 $Num\ a \Rightarrow a$, 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. How is it that we can write the constant 42, say, both in a context requiring an Int and in one requiring a $Float$ (say). Somehow Haskell "knows" which version of 42 is required in a given context. But, what is the type of 42 itself? The answer is that it has type $Num\ a \Rightarrow a$, 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 most 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 literal numbers such as 42 are actually considered to be shorthand for $fromInteger\ 42$. Since $fromInteger$ has type $Num\ a \Rightarrow Integer \rightarrow a$, then $fromInteger\ 42$ has type $Num\ a \Rightarrow a$.

The complete hierarchy of numeric classes is shown in Figure 7.2; 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 Appendix 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 7.3. The instance declarations implied by this table can be found in the Haskell Report.

7.4.2 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.

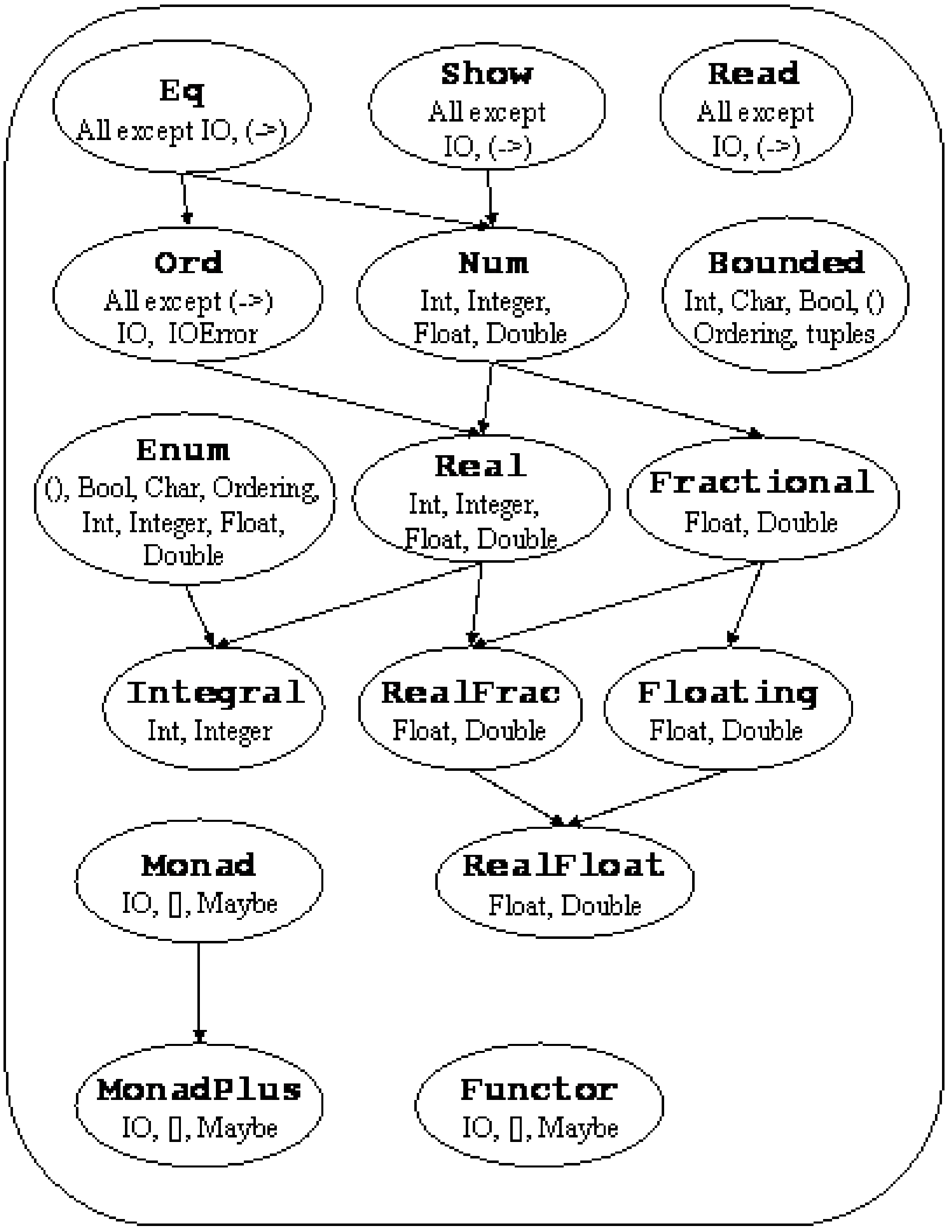


Figure 7.2: Numeric Class Hierarchy

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 7.3: Standard Numeric Types

Not all of the operations in the *Show* class will be discussed here, in fact the only one of interest is *show*:

```
class Show a where
  show :: a → String
  ...
```

Instances of *Show* can be derived, so normally we do not have to worry about the details of the definition of *show*.

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:

```
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![2]

```
main    = putStr (quine q)
quine s = s ++ show s
q       = "main = putStr (quine q)\nquine s = s ++ show s\nq = "
```

Details: The "\n" that appears twice in the string q is a "newline" character; that is, when q is printed (or displayed on a console) the string starting to the right of "\n" will appear on a new line.

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.

7.4.3 The Functor Class

[Define *Functor* class, and show instances for lists, *Maybe*, *Primitive*, *Music*, ...]

TBD

7.5 Other Derived Instances

In addition to *Eq* and *Ord*, instances of *Enum*, *Bounded*, *Ix*, *Read*, and *Show* (see Appendix B) can also be generated by the **deriving** clause. These 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 (i.e. values) of the type involved. For example, from:

```
data Color = Black
    | Blue
    | Green
```

[2] 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.

$\quad\quad | \; Cyan$
$\quad\quad | \; Red$
$\quad\quad | \; Magenta$
$\quad\quad | \; Yellow$
$\quad\quad | \; White$
$\quad$ **deriving** $(Show, Eq, Ord, Enum, Bounded)$

we can expect that:

$show\ [Red\ ..]$
$\Longrightarrow$ `"[Black,Blue,Green,Cyan,Red,Magenta,Yellow,White]"`

We can also expect that:

$minBound :: Color \Longrightarrow Black$
$maxBound :: Color \Longrightarrow White$

Note that the type signature "$::Color$" is given explicitly in this case, because, out of any context, at least, Haskell does not know the type for which you are trying to determine the minimum and maximum bounds.

Further details about derived instances can be found in the Haskell Report.

Many of the predefined data types in Haskell have **deriving** clauses, even ones with special syntax. For example, if we could write a data type declaration for lists (the reason we cannot do this is that lists have special syntax, both at the value and type level) it would look something like this:

data $[a] = [\,]$
$\quad\quad\quad | \;\; a : [a]$
$\quad$ **deriving** (Eq, Ord)

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, respectively (more on this later). An intransitive $(==)$ predicate, for example, would be problematic, confusing readers of the program who expect $(==)$ to be transitive. Nevertheless, it is sometimes necessary to provide Eq or Ord instances different from those that would be derived.

The data type declarations for *PitchClass*, *Primitive*, *Music* and *Control* given in Chapter 1 are not the ones actually used in Eutperpea. The actual definitions each use a **deriving** clause, and are shown in Figure 7.4. The *InstrumentName* data type from Chapter 1 also has a deriving clause for *Show*, *Eq*, and *Ord* (but is ommitted here to save space).

> **Details:** When instances of more than one type class are derived for the same data type, they appear grouped in parentheses as in Figure 7.4. Also, in this case *Eq* *must* appear if *Ord* does (unless an explicit instance for *Eq* is given), since *Eq* is a superclass of *Ord*.

Note that with single and double sharps and flats, there are many enharmonic equivalences. Thus in the data declaration for *PitchClass*, the constructors are ordered such that, if $pc_1 < pc_2$, then $pcToInt\ pc_1 \leqslant pcToInt\ pc_2$.

For some examples, the *Show* class allows us to convert values to strings:

$$\begin{array}{ll} show\ Cs & \Longrightarrow \texttt{"Cs"} \\ show\ concertA & \Longrightarrow \texttt{"(A,4)"} \end{array}$$

The *Read* class allows us to go the other way around:

$$\begin{array}{ll} read\ \texttt{"Cs"} & \Longrightarrow Cs \\ read\ \texttt{"(A,4)"} & \Longrightarrow (A, 4) \end{array}$$

The *Eq* class allows testing values for equality:

$$\begin{array}{ll} concertA == a440 & \Longrightarrow True \\ concertA == (A, 5) & \Longrightarrow False \end{array}$$

And the *Ord* class has relational operators for types whose values can be ordered:

$$\begin{array}{ll} C < G & \Longrightarrow True \\ max\ C\ G & \Longrightarrow G \end{array}$$

The *Enum* class is for “enummerable types.” For example:

$$\begin{array}{ll} succ\ C & \Longrightarrow Dff \\ pred\ 1 & \Longrightarrow 0 \\ fromEnum\ C & \Longrightarrow 2 \\ toEnum\ 3 & \Longrightarrow Dff \end{array}$$

The *Enum* class is also the basis on which *arithmetic sequences* (defined earlier in Section 5.3.1) are defined.

```
data PitchClass = Cff | Cf | C | Dff | Cs | Df | Css | D | Eff | Ds
                | Ef | Fff | Dss | E | Ff | Es | 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, Bounded)

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

data Music a =
      Prim (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)

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

Figure 7.4: Euterpea's Data Types with Deriving Clauses

7.6 The type of *play*

Ever since the *play* function was introduced in Chapter 2, we have been using it to "play" the results of our *Music* values, i.e. to listen to their rendering through MIDI. However, it is just a function like any other function in Haskell, but we never discussed what its type is. In fact, here it is:

$$play :: Performable\ a \Rightarrow Music\ a \rightarrow IO\ ()$$

The type of the result, *IO* (), is the type of a *command* in Haskell, i.e. something that "does I/O." We will have more to say about this in a later chapter.

But of more relevance to this chapter, note the constraint *Performable a*. You might guess that *Performable* is a type class, indeed it is the type class of "performable values." If a type is a member of (i.e. instance of) *Performable*, then it can be "performed," i.e. rendered as sound. The point is, some things we would not expect to be performable, for example a list or a character or a function. So the type signature for *play* can be read, "For any type T that is a member of the class *Performable*, *play* has type $Music\ T \rightarrow IO\ ()$."

Currently the types *Pitch*, (*Pitch*, *Volume*), and (*Pitch*, [*NoteAttribute*]) are members of the class *Performable*. (The *NoteAttribute* data type will be introduced in Chapter 8.) Indeed, we have used *play* on the first two of these types, i.e. on values of type *Music Pitch* and *Music* (*Pitch*, *Volume*) in previous examples, and you might have wondered how both could possibly be properly typed—hopefully now it is clear.

7.7 Reasoning With Type Classes

Type classes often imply a set of *laws* that govern the use of the operators in the class. For example, for the *Eq* class, we can expect the following laws to hold for every instance of the class:

$$\begin{array}{ll} x == x & \\ x == y & \supseteq y == x \\ (x == y) \wedge (y == z) & \supseteq x == z \\ (x \neq y) & \supseteq \neg\ (x == y) \end{array}$$

where $\supseteq$ should be read "implies that."[3]

[3]Mathematically, the first three of these laws are the same as those for an *equivalence relation*.

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 of interest for a particular 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 is defined, whether for our own type 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:

$$
\begin{array}{ll}
a \leqslant a & \\
(a \leqslant b) \wedge (b \leqslant c) & \supseteq (a \leqslant c) \\
(a \leqslant b) \wedge (b \leqslant a) & \supseteq (a == b) \\
(a \neq b) & \supseteq (a < b) \vee (b < a)
\end{array}
$$

Similar laws should hold for $(>)$.

But alas, the instance of *Music* in the class *Ord* given in Section 7.3.3 does not satisfy all of these laws! To see why, consider two *Primitive* values p_1 and p_2 such that $p_1 < p_2$. Now consider these two *Music* values:

$$
\begin{array}{l}
m_1 = \mathit{Prim}\ p_1 :+: \mathit{Prim}\ p_2 \\
m_2 = \mathit{Prim}\ p_2 :+: \mathit{Prim}\ p_1
\end{array}
$$

Clearly $m_1 == m_2$ is false, but the problem is, so are $m_1 < m_2$ and $m_2 < m_1$, thus violating the last law above.

To fix the problem, a *lexicographic ordering* should be used 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
  Prim p1       < Prim p2       = p1 < p2
  Prim p1       < _             = True
  (m1 :+: m2)   < Prim _        = False
  (m1 :+: m2)   < (m3 :+: m4)   = (m1 < m3) ||
                                  (m1 == m3) && (m2 < m4)
  (m1 :+: m2)   < _             = True
  (m1 :=: m2)   < Prim _        = False
  (m1 :=: m2)   < (m3 :+: m4)   = False
  (m1 :=: m2)   < (m3 :=: m4)   = (m1 < m3) ||
                                  (m1 == m3) && (m2 < m4)
```

$$
\begin{array}{lll}
(m_1 :=: m_2) & < \_ & = \mathit{True} \\
\mathit{Modify}\ c_1\ m_1 & < \mathit{Modify}\ c_2\ m_2 & = (c_1 < c_2) \vee \\
 & & \quad (c_1 == c_2) \wedge (m_1 < m_2) \\
\mathit{Modify}\ c_1\ m_1 & < \_ & = \mathit{False}
\end{array}
$$

This example shows the value of checking to be sure that each instance 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 7.1 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 7.2 Write out appropriate instance declarations for the *Color* type in the classes *Eq*, *Ord*, and *Enum*. (For simplicity you may define *Color* to have fewer constructors, say just *Red*, *Green* and *Blue*.)

Exercise 7.3 Define a type class called *Temporal* whose members are types that can be interpreted as having a temporal duration. *Temporal* should have three operations, namely:

$$
\begin{array}{ll}
\mathit{durT} & :: \mathit{Temporal}\ a \Rightarrow a \rightarrow \mathit{Dur} \\
\mathit{takeT} & :: \mathit{Temporal}\ a \Rightarrow \mathit{Dur} \rightarrow a \rightarrow a \\
\mathit{dropT} & :: \mathit{Temporal}\ a \Rightarrow \mathit{Dur} \rightarrow a \rightarrow a
\end{array}
$$

Then define instances of *Temporal* for the types *Music* and *Primitive*. (Hint: this is not as hard as it sounds, because you can *reuse* some function names previously defined to do these sorts of operations.)

Can you think of other types that are temporal?

Exercise 7.4 Functions are not members of the *Eq* class, because, in general, determining whether two functions are equal is undecideable. But functions whose domains are finite, and can be completely enumerated, *can* be tested for equality. We just need to test that each function, when applied to each element in the domain, returns the same result.

Define an instance of *Eq* for functions. For this to be possible, note that, if the function type is $a \rightarrow b$, then:

- the type a must be *enumerable* (i.e. a member of the *Enum* class),

- the type a must be *bounded* (i.e. a member of *Bounded* class), and
- the type b must admit *equality* (i.e. be a member of the *Eq* class).

These constraints must therefore be part of the instance declaration.

Hint: using the minimum and maximum bounds of a type, enumerate all the elements of that type using an arithmetic sequence (recall Section 5.3.1), which, despite its name, works for any enumerable type.

Test your implementation by defining some functions on existing Euterpea types that are finite and bounded (such as *PitchClass* and *Color*), or by defining some functions on your own data type(s).

Chapter 8

Interpretation and Performance

```
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}
module Euterpea.Music.Note.Performance where
import Euterpea.Music.Note.Music
import Euterpea.Music.Note.MoreMusic
```

> **Details:** The first line above is a GHC *pragma* that, in this case, relaxes certain constraints on instance declarations. Specifically, instances cannot normally be declared for type synonyms—but the above pragma overrides that constraint.

So far, our presentation of musical values in Haskell has been mostly structural, i.e. *syntactic*. Although we have given an interpretation of the duration of *Music* values (as manifested in *dur*, *takeM*, *dropM*, and so on), we have not given any deeper musical interpretation. 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 Euterpea—but still flexible, so that more than one interpretation is possible, just as in the human performance of music.

8.1 Abstract Performance

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 Haskell:

```
type Performance = [Event]

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

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

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 [Double]
  deriving (Show, Eq, Ord)
```

except that the former also defines "field labels" *eTime*, *eInst*, *ePitch*, *eDur*, *eVol*, and *eParams*, which can be used to create, update, and select from *Event* values.

Details: For example, this equation:

$$e = Event\ 0\ Cello\ 27\ (1/4)\ 50\ [\,]$$

is equivalent to:

$$\begin{aligned} e = Event\ \{ & eTime = 0, ePitch = 27, eDur = 1/4, \\ & eInst = Cello, eVol = 50, eParams = [\,]\} \end{aligned}$$

Although more verbose, the latter is also more descriptive, 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, using the value of e above, $eInst\ e \Rightarrow Cello$, $eDur\ e \Rightarrow 1/4$, 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.

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 $eParams$ of an event is for instruments other than MIDI, in particular instruments that we might design on our using the techniques described in Chapter 19.

An abstract performance is the lowest of our music representations not yet committed to MIDI or some other low-level computer music representation. In Chapter 14 we will discuss how to map a performance into MIDI.

8.1.1 Context

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, starting pitch offset, and tempo. We can think of this as the "context" in which a musical value is interpreted. This context can be captured formally in Haskell as a data type:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Context}\ a = \mathit{Context}\ \{\ \mathit{cTime} \quad :: \mathit{PTime}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{cPlayer} :: \mathit{Player}\ a, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{cInst} \quad :: \mathit{InstrumentName}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{cDur} \quad :: \mathit{DurT}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{cPch} \quad :: \mathit{AbsPitch}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{cVol} \quad :: \mathit{Volume}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{cKey} \quad :: (\mathit{PitchClass}, \mathit{Mode})\ \} \\
\quad \textbf{deriving}\ \mathit{Show}
\end{array}
$$

When a *Music* value is interpreted, it will be given an inital context, but as the *Music* value is recursively interpreted, the context will be updated to reflect things like tempo change, transposition, and so on. This will be made clear shortly.

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:

$$
\begin{array}{ll}
\mathit{metro} & :: \mathit{Int} \rightarrow \mathit{Dur} \rightarrow \mathit{DurT} \\
\mathit{metro}\ \mathit{setting}\ \mathit{dur} & = 60 / (\mathit{fromIntegral}\ \mathit{setting} * \mathit{dur})
\end{array}
$$

Thus, for example, $\mathit{metro}\ 96\ \mathit{qn}$ creates a tempo of 96 quarter notes per minute.

Details: $\mathit{fromIntegral} :: (\mathit{Integral}\ a, \mathit{Num}\ b) \Rightarrow a \rightarrow b$ coerces a value whose type is a member of the *Integral* class to a value whose type is a member of the *Num* class. As used here, it is effectively converting the *Int* value *setting* to a *Rational* value, because *dur* is a *Rational* value, *Rational* is a member of the *Num* class, and multiplication has type $(*) :: \mathit{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$.

8.1.2 Player Map

In addition to the context, we also need to know what *player* to use; that is, we need a mapping from each *PlayerName* (a string) in a *Music* value to the actual player to be used.[1] The details of what a player is, and how it gives great flexibility to Euterpea, will be explained later in this chapter

[1] We do not need a mapping from *InstrumentNames* to instruments, since that is handled in the translation from a performance into MIDI, which is discussed in Chapter 14.

```
perform :: PMap a → Context a → Music a → Performance
perform pm
  c@Context {cTime = t, cPlayer = pl, cDur = dt, cPch = k} m =
  case m of
    Prim (Note d p)            → playNote pl c d p
    Prim (Rest d)              → []
    m1 :+: m2                  →
      let c' = c {cTime = t + dur m1 ∗ dt}
      in perform pm c m1 ++ perform pm c' m2
    m1 :=: m2                  → merge (perform pm c m1)
                                       (perform pm c m2)
    Modify (Tempo r)         m → perform pm (c {cDur = dt / r})           m
    Modify (Transpose p)     m → perform pm (c {cPch = k + p})            m
    Modify (Instrument i)    m → perform pm (c {cInst = i})               m
    Modify (KeySig pc mo)    m → perform pm (c {cKey = (pc, mo)})         m
    Modify (Player pn)       m → perform pm (c {cPlayer = pm pn})         m
    Modify (Phrase pa)       m → interpPhrase pl pm c pa                  m
```

Figure 8.1: An abstract *perform* function

(Section 8.2). For now, we simply define a type synonym to capture the mapping of *PlayerName* to *Player*:

type *PMap a* = *PlayerName* → *Player a*

8.1.3 Interpretation

Finally, we are ready to give an interpretation to a piece of music, which we do by defining a function *perform*, whose type is:

perform :: *PMap a* → *Context a* → *Music a* → *Performance*

So *perform pm c m* is the *Performance* that results from interpreting *m* using player map *pm* in the initial context *c*. Conceptually, *perform* is perhaps the most important function defined in this textbook, and is shown in Figure 8.1. To help in understanding the definition of *perform*, let's step through the equations one at a time.

1. The interpretation of a note is player dependent. This is handled in *perform* using the *playNote* function, which takes the player as an

argument. Precisely how the *playNote* function works is described in Section 8.2, but for now you can think of it as returning a *Performance* (a list of events) with just one event: the note being played.

2. In the interpretation of (:+:), note that the *Performances* of the two arguments are appended together, with the start time of the second *Performance* delayed by the duration of the first (as captured in the context c'). The function *dur* (defined in Section 6.5) is used to compute this duration. Note that the interpretation of (:+:) is well-defined even for infinite *Music* values.

3. In the interpretation of (:=:), the *Performances* derived from the two arguments are merged into a time-ordered stream. The definition of *merge* is given below:

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

Note that *merge* is esssentially the same as the *mergeLD* function defined in Section 6.9.1.

4. In the interpretation of *Modify*, first recall the definition of *Control* from Chapter 2.2:

```
data Control =
      Tempo      Rational          -- scale the tempo
    | Transpose  AbsPitch          -- transposition
    | Instrument InstrumentName    -- instrument label
    | Phrase     [PhraseAttribute] -- phrase attributes
    | Player     PlayerName        -- player label
    | KeySig     PitchClass Mode   -- key signature and mode
  deriving (Show, Eq, Ord)
type PlayerName = String
data Mode       = Major | Minor
  deriving (Show, Eq, Ord)
```

Each of these six constructors is handled by a separate equation in the definition of *perform*. Note how the context is updated in each case—the *Context*, in general, is the running "state" of the performance, and gets updated in several different ways.

Also of note is the treatment of *Phrase*. Like the playing of a note, the playing of a phrase is player dependent. This is captured through the function *interpPhrase*, which takes the player as an argument. Like *playNote*, this too, along with the *PhraseAttribute* data type, will be described in full detail in Section 8.2.

Figure 8.2 is a block diagram showing how *perform* fits into the "big picture" of Euterpea. *Music* values are most abstract, *Performance* values are less abstract, and MIDI or audio streams are the least abstract. This chapter focuses on converting a *Music* value into a *Performance*; subsequent chapters will focus on translating a *Performance* into either MIDI (still at the "note" level, and fairly straightforward) or audio (at the "signal" level, and more complex).

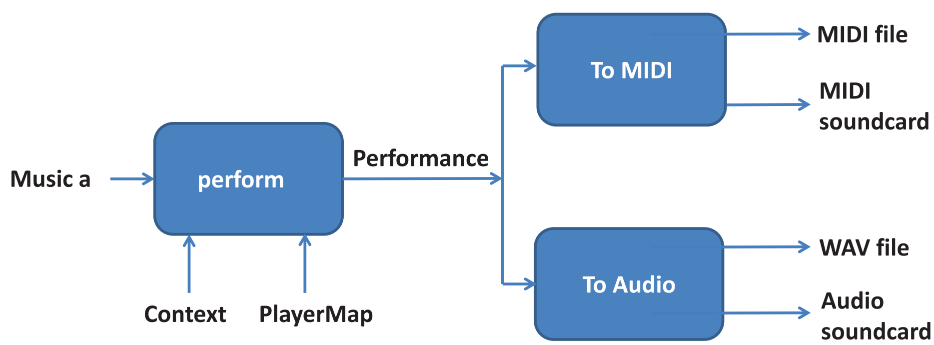


Figure 8.2: Block Diagram of Performance Concepts

8.1.4 Efficiency Concerns

The use of *dur* in the treatment of (:+:) can, in the worst case, result in a quadratic time complexity for *perform*. (Why?) 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 8.3.

Aside from efficiency, there is a more abstract reason for including duration in the result of *perform*. Namely, the performance of a rest is not just nothing—it is a period of "silence" equal in duration to that of the rest. Indeed, John Cage's famous composition *4' 33"*, in which the performer is instructed to play nothing, would otherwise be meaningless.[2]

Also 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 of *merge* that will work just as well in practice:

$$
\begin{array}{l}
\mathit{merge} :: \mathit{Performance} \rightarrow \mathit{Performance} \rightarrow \mathit{Performance} \\
\mathit{merge}\ [\,] \qquad\qquad \mathit{es}_2 \qquad\qquad = \mathit{es}_2 \\
\mathit{merge}\ \mathit{es}_1 \qquad\quad [\,] \qquad\qquad\ = \mathit{es}_1 \\
\mathit{merge}\ a@(e_1 : \mathit{es}_1)\ b@(e_2 : \mathit{es}_2) = \\
\quad \mathbf{if}\ \mathit{eTime}\ e_1 < \mathit{eTime}\ e_2\ \mathbf{then}\ e_1 : \mathit{merge}\ \mathit{es}_1\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mathbf{else}\ \ e_2 : \mathit{merge}\ a\ \mathit{es}_2
\end{array}
$$

8.2 Players

Recall from Section 2.2 that the *Phrase* constructor in the *Control* data type takes a list of *PhraseAttribute*s as an argument:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Control} = ... \\
\quad |\ \mathit{Phrase}\ [\mathit{PhraseAttribute}] \quad \text{-- phrase attributes} \\
\quad ...
\end{array}
$$

It is now time to unveil the definition of *PhraseAttribute*! Shown fully in Figure 8.4, 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 human players interpret things in different ways, of course, but even more fundamental is the fact that a pianist, for exam-

[2] In reality this piece is meant to capture extemporaneously the sound of the environment during that period of "silence." [Cag86]

```haskell
perform :: PMap a -> Context a -> Music a -> Performance
perform pm c m = fst (perf pm c m)

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

Figure 8.3: A more efficient *perform* function

ple, realizes legato in a way fundamentally different from the way a violinist does, because of differences in their instruments. Similarly, diminuendo on a piano and diminuendo on a harpsichord are very different concepts.

In addition to phrase attributes, Euterpea has a notion of *note attributes* that can similarly be interpreted in different ways by different players. This is done by exploiting polymorphism to define a version of *Music* that in addition to pitch, carries a list of note attributes for each individual note:

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

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

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

To facilitate the use of *Music1* values, Euterpea defines the following simple coercion functions:

```
toMusic1  :: Music Pitch → Music1
toMusic1  = mMap (λp → (p, []))
toMusic1' :: Music (Pitch, Volume) → Music1
toMusic1' = mMap (λ(p, v) → (p, [Volume v]))
```

Finally, with a slight stretch of the imagination, we can even consider the generation of a *score* as a kind of player: exactly how the music is notated 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?

To handle these three different kinds of interpretation, Euterpea has a notion of a *player* that "knows" about differences with respect to performance and notation. An Euterpean *Player* is a four-tuple consisting of a name and three functions: one for interpreting notes, one for phrases, and one for producing a properly notated score:

```
data PhraseAttribute = Dyn Dynamic
                     | Tmp Tempo
                     | Art Articulation
                     | Orn Ornament
  deriving (Show, Eq, Ord)
data Dynamic = Accent Rational | Crescendo Rational
             | Diminuendo Rational | StdLoudness StdLoudness
             | Loudness Rational
  deriving (Show, Eq, Ord)
data StdLoudness = PPP | PP | P | MP | SF | MF | NF | FF | FFF
  deriving (Show, Eq, Ord, Enum)
data Tempo = Ritardando Rational | Accelerando Rational
  deriving (Show, Eq, Ord)
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 (Show, Eq, Ord)
data Ornament = Trill | Mordent | InvMordent | DoubleMordent
              | Turn | TrilledTurn | ShortTrill
              | Arpeggio | ArpeggioUp | ArpeggioDown
              | Instruction String | Head NoteHead
              | DiatonicTrans Int
  deriving (Show, Eq, Ord)
data NoteHead = DiamondHead | SquareHead | XHead | TriangleHead
              | TremoloHead | SlashHead | ArtHarmonic | NoHead
  deriving (Show, Eq, Ord)
```

Figure 8.4: Phrase Attributes

```
data Player a = MkPlayer {pName        :: PlayerName,
                          playNote     :: NoteFun a,
                          interpPhrase :: PhraseFun a,
                          notatePlayer :: NotateFun a}
type NoteFun a   = Context a → Dur → a → Performance
type PhraseFun a = PMap a → Context a → [PhraseAttribute]
                     → Music a → (Performance, DurT)
type NotateFun a = ()
instance Show a ⇒ Show (Player a) where
  show p = "Player " ++ pName p
```

Note that *NotateFun* is just the unit type; this is because notation is currently not implemented in Euterpea. Also note the instance declaration for a *Player*—since its components are mostly functions, which are not default instances of *Show*, we define a simple way to return the *PlayerName*.

8.2.1 Example of Player Construction

In this section we define a "default player" called *defPlayer* (not to be confused with a "deaf player"!) for use when none other is specified in a score; it also functions as a basis from which other players can be derived.

At the upper-most level, *defPlayer* is defined as a four-tuple:

```
defPlayer :: Player Note1
defPlayer = MkPlayer
          {pName        = "Default",
           playNote     = defPlayNote     defNasHandler,
           interpPhrase = defInterpPhrase defPasHandler,
           notatePlayer = ()}
```

The remaining functions are defined in Figure 8.5. Before reading this code, first review how players are invoked by the *perform* function defined in the last section; in particular, note the calls to *playNote* and *interpPhrase*. We will define *defPlayer* to respond only to the *Volume* note attribute and to the *Accent*, *Staccato*, and *Legato* phrase attributes.

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 *defNasHandler*.

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

defNasHandler :: Context a → NoteAttribute → Event → Event
defNasHandler c (Volume v)     ev = ev {eVol = v}
defNasHandler c (Params pms) ev = ev {eParams = pms}
defNasHandler _ _              ev = ev

defInterpPhrase ::
  (PhraseAttribute → Performance → Performance) →
  (PMap a → Context a → [PhraseAttribute] →    -- PhraseFun
   Music a → (Performance, DurT))
defInterpPhrase pasHandler pm context pas m =
  let (pf, dur) = perf pm 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
```

Figure 8.5: Definition of default player *defPlayer*.

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 *Legato*). Thus *Modify* (*Phrase* [*Legato* (5/4)]) *m* effectively increases the duration of each note in *m* by 25% (without changing the tempo).

8.2.2 Deriving New Players From Old Ones

It should be clear that much of the code in Figure 8.5 can be re-used in defining a new player. For example, to define a player *newPlayer* that interprets note attributes just like *defPlayer* but behaves differently with respect to certain phrase attributes, we could write:

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

and then supply a suitable definition of *myPasHandler*. Better yet, we could just do this:

```
newPlayer :: Player (Pitch, [NoteAttribute])
newPlayer = defPlayer
  {pName        = "NewPlayer",
   interpPhrase = defInterpPhrase myPasHandler}
```

This version uses the "record update" syntax to directly derive the new player from *defPlayer*.

The definition of *myPasHandler* can 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
```

8.2.3 A Fancy Player

Figure 8.6 defines a more sophisticated player called *fancyPlayer* that knows all that *defPlayer* knows, and more. Note that *Slurred* is different from *Legato* in that it does not extend the duration of the *last* note(s). The behavior of *Ritardando x* can be explained as follows. We would 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*.

8.3 Putting it all Together

The *play* function in Euterpea ueses a default player map and a default context that are defined as follows:

```
defPMap           :: PMap Note1
defPMap "Fancy"   = fancyPlayer
defPMap "Default" = defPlayer
defPMap n         = defPlayer {pName = n}

defCon :: Context Note1
defCon = Context {cTime   = 0,
                  cPlayer = fancyPlayer,
                  cInst   = AcousticGrandPiano,
                  cDur    = metro 120 qn,
                  cPch    = 0,
                  cKey    = (C, Major),
                  cVol    = 127}
```

Note that if anything other than a `"Fancy"` or `"Default"` player is specified in the *Music* value, such as *player* `"Strange"` *m*, then the default player *defPlayer* is used, and given the name `"Strange"`.

If instead we wish to use our own player, say *newPlayer* defined in Section 8.2.2, then a new player map can be defined, such as:

```
myPMap             :: PlayerName -> Player Note1
myPMap "NewPlayer" = newPlayer
myPMap p           = defPMap p
```

Similarly, different versions of the context can be defined based on a user's needs.

We could, then, use these versions of player maps and contexts to invoke the *perform* function to generate an abstract *Performance*. Of course, we ultimately want to hear our music, not just see an abstract *Performance* displayed on our computer screen. Recall that *play*'s type signature is:

```
play :: Performable a => Music a -> IO ()
```

To allow using different player maps and contexts, Euterpea also has a version of *play* called *playA* whose type signature is:

```
playA :: Performable a =>
           PMap Note1 -> Context Note1 -> Music a -> IO ()
```

For example, to play a *Music* value *m* using *myPMap* defined above and the default context *defCon*, we can do:

playA myPMap defCon m

In later chapters we will learn more about *play*, and how it converts a *Performance* into MIDI events that eventually are heard through your computer's sound card.

Exercise 8.1 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 8.2 Choose some of the other phrase attributes and provide interpretations for them.

(Hint: As in *fancyPlayer*, you may not be able to use the "*pasHandler*" approach to implement some of the phrase attributes. For example, for a proper treatment of *Trill* (and similar ornaments) you will need to access the *cKey* field in the context.)

Exercise 8.3 Define a player *myPlayer* that appropriately handles the *Pedal* articulation and both the *ArpeggioUp* and *ArpeggioDown* ornamentations. You should define *myPlayer* as a derivative of *defPlayer* or *newPlayer*.

Exercise 8.4 Define a player *jazzMan* (or *jazzWoman* if you prefer) that plays a melody using a jazz "swing" feel. Since there are different kinds and degrees of swing, we can be more specific as follows: whenever there is a sequence of two eighth notes, they should be interpreted instead as a quarter note followed by an eighth note, but with tempo 3/2. So in essence, the first note is lengthened, and the second note is shortened, so that the first note is twice as long as the second, but they still take up the same amount of overall time.

(Hint: There are several ways to solve this problem. One surprisingly effective and straightforward solution is to implement *jazzMan* as a *NoteFun*, not a *PhraseFun*. In jazz, if an eighth note falls on a quarter-note beat it is said to fall on the "downbeat," and the eighth notes that are in between are said to fall on the "upbeat." For example, in the phrase *c* 4 *en* :+: *d* 4 *en* :+: *e* 4 *en* :+: *f* 4 *en*, the C and E fall on the downbeat, and the D and F fall on the upbeat. So to get a "swing feel," the notes on the down beat need to be lengthened, and ones on the upbeat need to be delayed and shortened. Whether an event falls on a downbeat or upbeat can be determined from the *cTime* and *cDur* of the context.)

Exercise 8.5 Implement the ornamentation *DiatonicTrans*, which is intended to be a "diatonic tranposition" of a phrase within a particular key. The argument to *DiatonicTrans* is an integer representing the number of *scale degrees* to do the transposition. For example, the diatonic transposition of $c\ 4\ en$:+: $d\ 4\ en$:+: $e\ 4\ en$ in C major by 2 scale degrees should yield $e\ 4\ en$:+: $f\ 4\ en$:+: $g\ 4\ en$, whereas in G major should yield $e\ 4\ en$:+: $fs\ 4\ en$:+: $g\ 4\ en$.

(Hint: You will need to access the key from the context (using *cKey*). Thus, as with *fancyPlayer*, you may not be able to use the "*pasHandler*" approach to solve this problem.)

```haskell
fancyPlayer :: Player (Pitch, [NoteAttribute])
fancyPlayer = MkPlayer {pName        = "Fancy",
                        playNote     = defPlayNote defNasHandler,
                        interpPhrase = fancyInterpPhrase,
                        notatePlayer = ()}

fancyInterpPhrase              :: PhraseFun a
fancyInterpPhrase pm c [] m = perf pm c m
fancyInterpPhrase pm
  c@Context {cTime = t, cPlayer = pl, cInst = i,
             cDur = dt, cPch = k, cVol = v}
  (pa : pas) m =
  let pfd@(pf, dur) = fancyInterpPhrase pm c pas m
      loud x        = fancyInterpPhrase pm 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 ((1 + (t - t0) * r) *
                                         fromIntegral v)}
                      in (map upd pf, dur)
  in case pa of
    Dyn (Accent x) ->
      (map (\e -> 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 pm
                              c {cVol = round 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 (\e -> e {eDur = x * eDur e}) pf, dur)
    Art (Legato x)       -> (map (\e -> e {eDur = x * eDur e}) pf, dur)
    Art (Slurred x)      ->
      let lastStartTime = foldr (\e 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
```

Figure 8.6: Definition of Player *fancyPlayer*.

Chapter 9

Self-Similar Music

```
module Euterpea.Examples.SelfSimilar where
import Euterpea
```

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 well-known 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.

9.1 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 9.1 shows the result of applying this process once to a four-note melody (the first four notes form the original 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

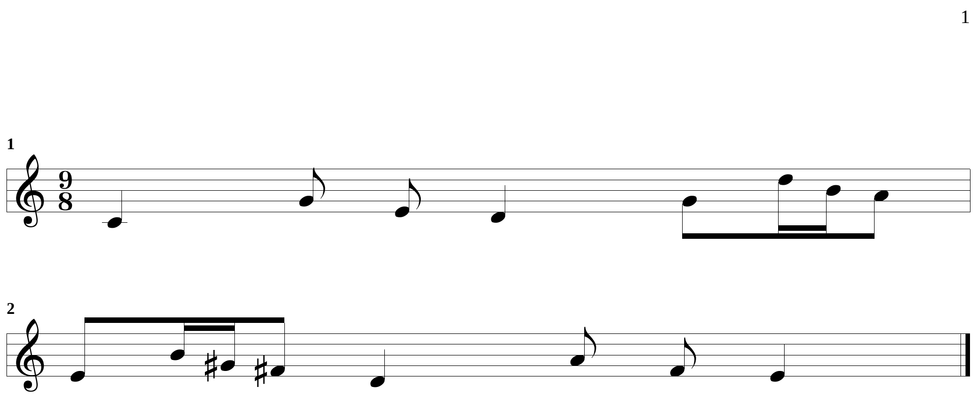


Figure 9.1: An Example of Self-Similar Music

level.

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

```
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 9.1), 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. This code should be studied carefully. In particualr, the recursion in *mkCluster* is different from what we have seen before, as it is not a direct invocation of *mkCluster*, but rather it is a high-order argument to *map* (which in turn invokes *mkCluster* an aribtrary number of times).

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

$$
\begin{aligned}
&\mathit{fringe} && :: \mathit{Int} \to \mathit{Cluster} \to [\mathit{SNote}] \\
&\mathit{fringe}\ 0\ (\mathit{Cluster}\ \mathit{note}\ \mathit{cls}) && = [\mathit{note}] \\
&\mathit{fringe}\ n\ (\mathit{Cluster}\ \mathit{note}\ \mathit{cls}) && = \mathit{concatMap}\ (\mathit{fringe}\ (n-1))\ \mathit{cls}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{concatMap} && :: (a \to [b]) \to [a] \to [b] \\
&\mathit{concatMap}\ f && = \mathit{concat} \circ \mathit{map}\ f
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
&\mathit{simToMusic} && :: [\mathit{SNote}] \to \mathit{Music}\ \mathit{Pitch} \\
&\mathit{simToMusic}\ \mathit{ss} && = \mathbf{let}\ \mathit{mkNote}\ (d, \mathit{ap}) = \mathit{note}\ d\ (\mathit{pitch}\ \mathit{ap}) \\
& && \quad\ \mathbf{in}\ \mathit{line}\ (\mathit{map}\ \mathit{mkNote}\ \mathit{ss})
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{simToMusic} && :: [\mathit{SNote}] \to \mathit{Music}\ \mathit{Pitch} \\
&\mathit{simToMusic} && = \mathit{line} \circ \mathit{map}\ \mathit{mkNote} \\
&\mathit{mkNote} && :: (\mathit{Dur}, \mathit{AbsPitch}) \to \mathit{Music}\ \mathit{Pitch} \\
&\mathit{mkNote}\ (d, \mathit{ap}) && = \mathit{note}\ d\ (\mathit{pitch}\ \mathit{ap})
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathit{ss}\ \mathit{pat}\ n\ \mathit{tr}\ \mathit{te} = \\
&\quad \mathit{transpose}\ \mathit{tr}\ \$\ \mathit{tempo}\ \mathit{te}\ \$\ \mathit{simToMusic}\ \$\ \mathit{fringe}\ n\ \$\ \mathit{selfSim}\ \mathit{pat}
\end{aligned}
$$

9.1.1 Sample Compositions

Let's start with a melody with no rhythmic variation.

$$
\begin{aligned}
&m_0 \;\; :: [\mathit{SNote}]\\
&m_0 \;\; = [(1,2),(1,0),(1,5),(1,7)]\\
&tm_0 = \mathit{instrument}\ \mathit{Vibraphone}\ (ss\ m_0\ 4\ 50\ 20)
\end{aligned}
$$

One fun thing to do with music like this is to combine it with variations of itself. For example:

$$ttm_0 = tm_0 :=: \mathit{transpose}\ (12)\ (\mathit{revM}\ \ tm_0)$$

We could also try the opposite: a simple percussion instrument with no melodic variation, i.e. all rhythm:

$$
\begin{aligned}
&m_1 \;\; :: [\mathit{SNote}]\\
&m_1 \;\; = [(1,0),(0.5,0),(0.5,0)]\\
&tm_1 = \mathit{instrument}\ \mathit{Percussion}\ (ss\ m_1\ 4\ 43\ 2)
\end{aligned}
$$

Note that the pitch is transposed by 43, which is the MIDI Key number for a "high floor tom" (i.e. percussion sound *HighFloorTom*—recall the discussion in Section 6.12).

Here is a very simple melody, two different pitches and two different durations:

$$
\begin{aligned}
&m_2 \;\; :: [\mathit{SNote}]\\
&m_2 \;\; = [(dqn,0),(qn,4)]\\
&tm_2 = ss\ m_2\ 6\ 50\ (1/50)
\end{aligned}
$$

Here are some more exotic compositions, combining both melody and rhythm:

$$
\begin{aligned}
&m_3 \;\;\; :: [\mathit{SNote}]\\
&m_3 \;\;\; = [(hn,3),(qn,4),(qn,0),(hn,6)]\\
&tm_3 \;\; = ss\ m_3\ 4\ 50\ (1/4)\\
&ttm_3 = \mathbf{let}\ l_1 = \mathit{instrument}\ \mathit{Flute}\ tm_3\\
&\qquad\qquad\quad l_2 = \mathit{instrument}\ \mathit{AcousticBass}\ \$\\
&\qquad\qquad\qquad\qquad \mathit{transpose}\ (-9)\ (\mathit{revM}\ \ tm_3)\\
&\qquad\quad\ \mathbf{in}\ l_1 :=: l_2\\
&m_4 \;\;\; :: [\mathit{SNote}]\\
&m_4 \;\;\; = [(hn,3),(hn,8),(hn,22),(qn,4),(qn,7),(qn,21),\\
&\qquad\quad\ (qn,0),(qn,5),(qn,15),(wn,6),(wn,9),(wn,19)]\\
&tm_4 \;\; = ss\ m_4\ 3\ 50\ 8
\end{aligned}
$$

Exercise 9.1 Experiment with this idea futher, using other melodic seeds, exploring different depths of the clusters, and so on.

Exercise 9.2 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.

9.2 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:

$$\begin{aligned}
&fringe' && :: Int \rightarrow Cluster \rightarrow [[SNote]]\\
&fringe'\ 0\ (Cluster\ note\ cls) && = [[note]]\\
&fringe'\ n\ (Cluster\ note\ cls) && = map\ (fringe\ (n-1))\ cls
\end{aligned}$$

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:

$$\begin{aligned}
&simToMusic' :: [[SNote]] \rightarrow Music\ Pitch\\
&simToMusic' = chord \circ map\ (line \circ map\ mkNote)
\end{aligned}$$

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

$$\begin{aligned}
&ss'\ pat\ n\ tr\ te =\\
&\quad transpose\ tr\ \$\ tempo\ te\ \$\ simToMusic'\ \$\ fringe'\ n\ \$\ selfSim\ pat
\end{aligned}$$

Using some of the same patterns used earlier, here are some sample compositions (with not necessarily a great outcome...):

$ss_1 = ss'\ m_2\ 4\ 50\ (1/8)$
$ss_2 = ss'\ m_3\ 4\ 50\ (1/2)$
$ss_3 = ss'\ m_4\ 3\ 50\ 2$

Here is a new one, based on a major triad:

$m_5 = [(en, 4), (sn, 7), (en, 0)]$
$ss_5 = ss\ m_5\ 4\ 45\ (1/500)$
$ss_6 = ss'\ m_5\ 4\ 45\ (1/1000)$

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

9.3 Other Self-Similar Structures

The reader will observe that our notion of "self-similar harmony" does 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 do is interpret the result differently. This is a common characteristic of algorithmic music composition—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. If alligned properly in time there will be a harmonic relationship between the levels, which could yield pleasing results.

The *Cluster* data type is conceptually useful in that is represents the infinite solution space of self-simlar melodies. And it is computationally useful in that it is computed to a desired depth only once, and thus can be inspected and reused without recomputing each level of the tree. This idea might be useful in the application mentioned above, namely combining two or more levels of the result in interesting ways.

However, the *Cluster* data type is strictly unnecessary, in that, for example, if we are interested in computing a specific level, we could define a function that recursed to that level and gave the result directly, without saving the intermediate levels.

A final point about the notion of self-similarity captured in this chapter is that the initial pattern is used as the basis with which to transform each successive level. Another strategy would be to use the entirety of each new level as the seed for transforming itself into the next level. This will result in an exponential blow-up in the size of each level, but may be worth pursuing—in some sense it is a simpler notion of self-similarity than what we have used in this chapter.

All of the ideas in this section, and others, we leave as exercises for the reader.

Exercise 9.3 Experiment with the self-similar programs in this chapter. Compose an interesting piece of music through a judicious choice of starting melody, depth of recursion, instrumentation, etc.

Exercise 9.4 Devise an interpretation of a *Cluster* that plays multiple levels of the *Cluster* in parallel. Try to get the levels to align properly in time so that each level has the same duration. You may choose to play all the levels up to a certain depth in parallel, or levels within a certain range, say levels 3 through 5.

Exercise 9.5 Define an alternative version of *simToMusic* that interprets the music differently. For example:

- Interpret the pitch as an index into a scale—e.g., as an index into the C major scale, so that 0 corresponds to C, 1 to D, 2 to E, 3 to F, ..., 6 to B, 7 to C in the next octave, and so on.
- Interpret the pitch as duration, and the duration as pitch.

Exercise 9.6 Modify the self-similar code in the following ways:

- Add a Volume component to *SNote* (in other words, define it as a triple instead of a pair), and redefine *addMult* so that it takes two of these triples and combines them in a suitable way. Then modify the rest of the code so that the result is a *Music1* value. With these modifications, compose something interesting that highlights the changes in volume.
- Change the *AbsPitch* field in *SNote* to be a list of *AbsPitch*s, to be interpreted ultimately as a chord. Figure out some way to combine them in *addMult*, and compose something interesting.

Exercise 9.7 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.

Exercise 9.8 Define a function that gives the same result as *ss*, but without using a data type such as *Cluster*.

Exercise 9.9 Define a version of self-similarity similar to that defined in this chapter, but that uses the entire melody generated at one level to transform itself into the next level (rather than using the original seed pattern).

Chapter 10

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.

10.1 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 is also true for length 1. But if it is 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 case*).
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*).

10.2 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 case by proving the property for length 0; i.e. for $xs = [\,]$:

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

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:

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

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 case by proving the property for length 0; i.e. for $xs = [\,]$:

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

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:

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

$$
\begin{array}{l}
\mathit{length}\ (x : (\mathit{xs} +\!\!+\ \mathit{ys})) \\
\ \Rightarrow \{\mathit{unfold\ length}\} \\
1 + \mathit{length}\ (\mathit{xs} +\!\!+\ \mathit{ys}) \\
\ \Rightarrow \{\mathit{induction\ hypothesis}\} \\
1 + (\mathit{length}\ \mathit{xs} + \mathit{length}\ \mathit{ys}) \\
\ \Rightarrow \{\mathit{associativity}\ \mathbf{of}\ (+)\} \\
(1 + \mathit{length}\ \mathit{xs}) + \mathit{length}\ \mathit{ys} \\
\ \Rightarrow \{\mathit{fold\ length}\} \\
\mathit{length}\ (x : \mathit{xs}) + \mathit{length}\ \mathit{ys}
\end{array}
$$

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

10.3 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 recursively defined versions. In particular, recall these two definitions of *toAbsPitches*:

$$
\begin{array}{ll}
\mathit{toAbsPitches}_1\ [\,] & = [\,] \\
\mathit{toAbsPitches}_1\ (p : \mathit{ps}) & = \mathit{absPitch}\ p : \mathit{toAbsPitches}_1\ \mathit{ps}
\end{array}
$$

$$\mathit{toAbsPitches}_2 = \mathit{map}\ \mathit{absPitch}$$

We want to prove that $\mathit{toAbsPitches}_1 = \mathit{toAbsPitches}_2$. To do so, we use the extensionality principle (briefly discussed in Section 3.6.1), which says that two functions are equal if, when applied to the same value, they always yield the same result. We can change the specification slightly to reflect this. For any finite list *ps*, we want to prove:

$$\mathit{toAbsPitches}_1\ \mathit{ps} = \mathit{toAbsPitches}_2\ \mathit{ps}$$

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

$$
\begin{array}{l}
\mathit{toAbsPitches}_1\ [\,] \\
\ \Rightarrow [\,] \\
\ \Rightarrow \mathit{map}\ \mathit{absPitch}\ [\,] \\
\ \Rightarrow \mathit{toAbsPitches}_2\ [\,]
\end{array}
$$

Next we assume that $\mathit{toAbsPitches}_1\ \mathit{ps} = \mathit{toAbsPitches}_2\ \mathit{ps}$ holds, and try to prove that $\mathit{toAbsPitches}_1\ (p : \mathit{ps}) = \mathit{toAbsPitches}_2\ (p : \mathit{ps})$:

$$
\begin{array}{l}
\mathit{toAbsPitches}_1\ (p : \mathit{ps}) \\
\ \Rightarrow \mathit{absPitch}\ p : \mathit{toAbsPitches}_1\ \mathit{ps}
\end{array}
$$

$\Rightarrow$ *absPitch* p : *toAbsPitches*$_2$ *ps*
$\Rightarrow$ *absPitch* p : *map absPitch ps*
$\Rightarrow$ *map absPitch* $(p : ps)$

Note the use of the induction hypothesis in the second step.

For a proof involving *foldr*, recall from Chapter 3 this recursive definition of *line*:

$$\begin{aligned} line_1\ [\,] &= rest\ 0 \\ line_1\ (m : ms) &= m \mathbin{:+:} line_1\ ms \end{aligned}$$

and this non-recursive version:

$$line_2 = foldr\ (:+:)\ (rest\ 0)$$

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

$line_1\ [\,]$
$\Rightarrow rest\ 0$
$\Rightarrow foldr\ (:+:)\ (rest\ 0)\ [\,]$
$\Rightarrow line_2\ [\,]$

Then the induction step:

$line_1\ (m : ms)$
$\Rightarrow m \mathbin{:+:} line_1\ ms$
$\Rightarrow m \mathbin{:+:} line_2\ ms$
$\Rightarrow m \mathbin{:+:} foldr\ (:+:)\ (rest\ 0)\ ms$
$\Rightarrow foldr\ (:+:)\ (rest\ 0)\ (m : ms)$
$\Rightarrow line_2\ (m : ms)$

The proofs of equivalence of the definitions of *toPitches*, *chord*, *maxPitch*, and *hList* from Chapter 3 are similar, and left as an exercise.

Exercise 10.1 From Chapter 3, prove that the original recursive versions of the following functions are equivalent to the versions using *map* or *fold*: *toPitches*, *chord*, *maxPitch*, and *hList*.

10.3.1 [Advanced] Reverse

The proofs of function equivalence in the last section were fairly straightforward. For something more challenging, consider the definition of *reverse* given in Section 3.5:

$$\begin{aligned}
&\mathit{reverse}_1\ [\,] &&= [\,]\\
&\mathit{reverse}_1\ (x : \mathit{xs}) &&= \mathit{reverse}_1\ \mathit{xs} +\!\!+ [x]
\end{aligned}$$

and the version given in Section 3.6:

$$\mathit{reverse}_2\ \mathit{xs} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ \mathit{xs}$$

We would like to show that these are the same; i.e. that $\mathit{reverse}_1\ \mathit{xs} = \mathit{reverse}_2\ \mathit{xs}$ for any finite list xs. In carrying out this proof one new idea will be demonstrated, namely the need for an *auxiliary property* which is proved independently of the main result.

The base case is easy, as it often is:

$$\begin{aligned}
&\mathit{reverse}_1\ [\,]\\
&\Rightarrow [\,]\\
&\Rightarrow \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ [\,]\\
&\Rightarrow \mathit{reverse}_2\ [\,]
\end{aligned}$$

Assume now that $\mathit{reverse}_1\ \mathit{xs} = \mathit{reverse}_2\ \mathit{xs}$. The induction step proceeds as follows:

$$\begin{aligned}
&\mathit{reverse}_1\ (x : \mathit{xs})\\
&\Rightarrow \mathit{reverse}_1\ \mathit{xs} +\!\!+ [x]\\
&\Rightarrow \mathit{reverse}_2\ \mathit{xs} +\!\!+ [x]\\
&\Rightarrow \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ \mathit{xs} +\!\!+ [x]\\
&\Rightarrow\ ???
\end{aligned}$$

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

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

in which case we could complete the proof as follows:

$$\begin{aligned}
&...\\
&\Rightarrow \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ \mathit{xs} +\!\!+ [x]\\
&\Rightarrow \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ (x : \mathit{xs})\\
&\Rightarrow \mathit{reverse}_2\ (x : \mathit{xs})
\end{aligned}$$

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:

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

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

$$\begin{array}{l} \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ \mathit{xs} +\!\!+ [x] \\ \ \Rightarrow \{\mathit{property}\ (2)\} \\ \mathit{foldl}\ (\mathit{flip}\ (:))\ ([\,] +\!\!+ [x])\ \mathit{xs} \\ \ \Rightarrow \{\mathit{unfold}\ (+\!\!+)\} \\ \mathit{foldl}\ (\mathit{flip}\ (:))\ [x]\ \mathit{xs} \\ \ \Rightarrow \{\mathit{fold}\ (\mathit{flip}\ (:))\} \\ \mathit{foldl}\ (\mathit{flip}\ (:))\ (\mathit{flip}\ (:)\ [\,]\ x)\ \mathit{xs} \\ \ \Rightarrow \{\mathit{fold}\ \mathit{foldl}\} \\ \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ (x : \mathit{xs}) \end{array}$$

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:

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

and the induction step proceeds as follows:

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

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

10.4 Useful Properties on Lists

There are many useful properties of functions on lists that require inductive proofs. Figures 10.1 and 10.2 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 $\mathit{reverse}_1$ and $\mathit{reverse}_2$ are the same; see if you can find them![1]

(Note, by the way, that in the first rule for *map* in Figure 10.1, 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.)

10.4.1 [Advanced] Function Strictness

Note that the last rule for *map* in Figure 10.1 is only valid for *strict* functions. A function f is said to be strict if $f\ \perp = \perp$. Recall from Section 1.4 that $\perp$ is the value associated with a non-terminating computation. So another 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 $(+1)\ \perp = \perp + 1 = \perp$. 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:

$$\mathit{const}\ x\ y = x$$

is strict on its first argument (why?), but not its second, because $\mathit{const}\ x\ \perp = x$, for any x.

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

Properties of map:

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

For all strict f:

$$f \circ head = head \circ map\ f$$

Properties of the $fold$ functions:

1. If op is associative, and e ‘op‘ $x = x$ and x ‘op‘ $e = x$ for all x, then for all finite xs:

   $$foldr\ op\ e\ xs = foldl\ op\ e\ xs$$

2. If the following are true:

   $$
   \begin{aligned}
   &x\ \text{‘}op_1\text{‘}\ (y\ \text{‘}op2\text{‘}\ z) &&= (x\ \text{‘}op_1\text{‘}\ y)\ \text{‘}op2\text{‘}\ z\\
   &x\ \text{‘}op_1\text{‘}\ e &&= e\ \text{‘}op2\text{‘}\ x
   \end{aligned}
   $$

   then for all finite xs:

   $$foldr\ op_1\ e\ xs = foldl\ op2\ e\ xs$$

3. For all finite xs:

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

Figure 10.1: Some Useful Properties of map and $fold$.

Properties of $(+\!\!+)$:

For all xs, ys, and zs:

$$\begin{aligned}(xs +\!\!+ ys) +\!\!+ zs &= xs +\!\!+ (ys +\!\!+ zs)\\ xs +\!\!+ [\,] &= [\,] +\!\!+ xs = xs\end{aligned}$$

Properties of *take* and *drop*:

$$\begin{aligned}\mathit{take}\ m \circ \mathit{take}\ n &= \mathit{take}\ (\mathit{min}\ m\ n)\\ \mathit{drop}\ m \circ \mathit{drop}\ n &= \mathit{drop}\ (m+n)\\ \mathit{take}\ m \circ \mathit{drop}\ n &= \mathit{drop}\ n \circ \mathit{take}\ (m+n)\end{aligned}$$

For all non-negative m and n such that $n \geqslant m$:

$$\mathit{drop}\ m \circ \mathit{take}\ n = \mathit{take}\ (n-m) \circ \mathit{drop}\ m$$

For all non-negative m and n, and finite xs:

$$\mathit{take}\ n\ xs +\!\!+ \mathit{drop}\ n\ xs = xs$$

Properties of *reverse*:

For all finite xs:

$$\begin{aligned}\mathit{reverse}\ (\mathit{reverse}\ xs) &= xs\\ \mathit{head}\ (\mathit{reverse}\ xs) &= \mathit{last}\ xs\\ \mathit{last}\ (\mathit{reverse}\ xs) &= \mathit{head}\ xs\end{aligned}$$

Figure 10.2: Useful Properties of Other Functions Over Lists

Details: Understanding strictness requires a careful understanding of Haskell's pattern-matching rules. For example, consider the definition of $(\wedge)$ from the Standard Prelude:

$$\begin{aligned}&(\wedge) && :: Bool \rightarrow Bool \rightarrow Bool\\ &True \wedge x && = x\\ &False \wedge \_ && = False\end{aligned}$$

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:

$$\begin{aligned}&f\ (head\ [\,])\\ &\Rightarrow f\ \bot\end{aligned}$$

$head\ [\,]$ 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:

$$\begin{aligned}&head\ (map\ f\ [\,])\\ &\Rightarrow head\ [\,]\\ &\Rightarrow \bot\end{aligned}$$

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:

$$\begin{aligned}&f\ (head\ (x : xs))\\ &\Rightarrow f\ x\end{aligned}$$

$\Rightarrow head\ (f\ x : map\ f\ xs)$
$\Rightarrow head\ (map\ f\ (x : xs))$

and we are done.

Exercise 10.2 Prove as many of the properties in Figures 10.1 and 10.2 as you can.

Exercise 10.3 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*, *dur*, *revM*, $(\wedge)$, $(True\ \wedge)$, $(False\ \wedge)$, and the following function:

$$\begin{array}{ll} ifFun & :: Bool \to a \to a \to a \\ ifFun\ pred\ cons\ alt = & \textbf{if}\ pred\ \textbf{then}\ cons\ \textbf{else}\ alt \end{array}$$

10.5 Induction on the Music Data Type

Proof by induction is not limited to lists. In particular, we can use it to reason about *Music* values.

For example, recall this property intuitively conjectured in Section 6.14:

$mFold\ Prim\ (:+:)\ (:=:)\ Modify\ m = m$

To prove this, we again use the extensionality principle, and then proceed by induction. But what is the base case? Recall that the *Music* data type is defined as:

```
data Music a =
      Prim (Primitive a)
    | Music a :+: Music a
    | Music a :=: Music a
    | Modify Control (Music a)
```

The only constructor that does not take a *Music* value as an argument is *Prim*, so that in fact is the only base case.

So, starting with this base case:

$mFold\ Prim\ (:+:)\ (:=:)\ Modify\ (Prim\ p)$
$\Rightarrow Prim\ p$
$\Rightarrow id\ (Prim\ p)$

That was easy! Next, we develop an induction step for each of the three non-base cases:

$$\begin{array}{l} \textit{mFold Prim } (:+:)\ (:=:)\ \textit{Modify } (m_1 :+: m_2) \\ \Rightarrow \textit{mFold Prim } (:+:)\ (:=:)\ \textit{Modify } m_1 :+: \\ \quad\ \textit{mFold Prim } (:+:)\ (:=:)\ \textit{Modify } m_2 \\ \Rightarrow m_1 :+: m_2 \\ \Rightarrow \textit{id } (m_1 :+: m_2) \end{array}$$

$$\begin{array}{l} \textit{mFold Prim } (:+:)\ (:=:)\ \textit{Modify } (m_1 :=: m_2) \\ \Rightarrow \textit{mFold Prim } (:+:)\ (:=:)\ \textit{Modify } m_1 :=: \\ \quad\ \textit{mFold Prim } (:+:)\ (:=:)\ \textit{Modify } m_2 \\ \Rightarrow m_1 :=: m_2 \\ \Rightarrow \textit{id } (m_1 :=: m_2) \end{array}$$

$$\begin{array}{l} \textit{mFold Prim } (:+:)\ (:=:)\ \textit{Modify } (\textit{Modify } c\ m) \\ \Rightarrow \textit{Modify } c\ (\textit{mFold Prim } (:+:)\ (:=:)\ \textit{Modify } m) \\ \Rightarrow \textit{Modify } c\ m \\ \Rightarrow \textit{id } (\textit{Modify } c\ m) \end{array}$$

These three steps were quite easy as well, but is not something we could have done without induction.

For something more challenging, let's consider the following:

$$\textit{dur } (\textit{revM } m) = \textit{dur } m$$

Again we proceed by induction, starting with the base case:

$$\begin{array}{l} \textit{dur } (\textit{revM } (\textit{Prim } p)) \\ \Rightarrow \textit{dur } (\textit{Prim } p) \end{array}$$

Sequential composition is straightforward:

$$\begin{array}{l} \textit{dur } (\textit{revM } (m_1 :+: m_2)) \\ \Rightarrow \textit{dur } (\textit{revM } m_2 :+: \textit{revM } m_1) \\ \Rightarrow \textit{dur } (\textit{revM } m_2) + \textit{dur } (\textit{revM } m_1) \\ \Rightarrow \textit{dur } m_2 + \textit{dur } m_1 \\ \Rightarrow \textit{dur } m_1 + \textit{dur } m_2 \\ \Rightarrow \textit{dur } (m_1 :+: m_2) \end{array}$$

But things get more complex with parallel composition:

$$\begin{array}{l} \textit{dur } (\textit{revM } (m_1 :=: m_2)) \\ \Rightarrow \textit{dur } (\textbf{let } d_1 = \textit{dur } m_1 \\ \qquad\qquad\quad d_2 = \textit{dur } m_2 \\ \qquad\quad \textbf{in if } d_1 > d_2 \textbf{ then } \textit{revM } m_1 :=: (\textit{rest } (d_1 - d_2) :+: \textit{revM } m_2) \\ \qquad\qquad\qquad\qquad\ \textbf{else } (\textit{rest } (d_2 - d_1) :+: \textit{revM } m_1) :=: \textit{revM } m_2) \end{array}$$

$$
\begin{aligned}
&\Rightarrow \textbf{let } d_1 = \mathit{dur}\ m_1 \\
&\qquad\quad d_2 = \mathit{dur}\ m_2 \\
&\quad\ \textbf{in if } d_1 > d_2 \textbf{ then } \mathit{dur}\ (\mathit{revM}\ m_1 :=: (\mathit{rest}\ (d_1 - d_2) :+: \mathit{revM}\ m_2)) \\
&\qquad\qquad\qquad\quad \textbf{else } \mathit{dur}\ ((\mathit{rest}\ (d_2 - d_1) :+: \mathit{revM}\ m_1) :=: \mathit{revM}\ m_2) \\
&\ldots
\end{aligned}
$$

At this point, to make things easier to understand, we will consider each branch of the conditional in turn. First the consequent branch:

$$
\begin{aligned}
&\mathit{dur}\ (\mathit{revM}\ m_1 :=: (\mathit{rest}\ (d_1 - d_2) :+: \mathit{revM}\ m_2)) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ (\mathit{revM}\ m_1))\ (\mathit{dur}\ (\mathit{rest}\ (d_1 - d_2) :+: \mathit{revM}\ m_2)) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ m_1)\ (\mathit{dur}\ (\mathit{rest}\ (d_1 - d_2) :+: \mathit{revM}\ m_2)) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ m_1)\ (\mathit{dur}\ (\mathit{rest}\ (d_1 - d_2)) + \mathit{dur}\ (\mathit{revM}\ m_2)) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ m_1)\ ((d_1 - d_2) + \mathit{dur}\ m_2) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ m_1)\ (\mathit{dur}\ m_1) \\
&\Rightarrow \mathit{dur}\ m_1
\end{aligned}
$$

And then the alternative:

$$
\begin{aligned}
&\mathit{dur}\ ((\mathit{rest}\ (d_2 - d_1) :+: \mathit{revM}\ m_1) :=: \mathit{revM}\ m_2) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ ((\mathit{rest}\ (d_2 - d_1) :+: \mathit{revM}\ m_1))\ (\mathit{dur}\ (\mathit{revM}\ m_2)) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ ((\mathit{rest}\ (d_2 - d_1) :+: \mathit{revM}\ m_1))\ (\mathit{dur}\ m_2) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ (\mathit{rest}\ (d_2 - d_1)) + \mathit{dur}\ (\mathit{revM}\ m_1))\ (\mathit{dur}\ m_2) \\
&\Rightarrow \mathit{max}\ ((d_2 - d_1) + \mathit{dur}\ m_1)\ (\mathit{dur}\ m_2) \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ m_2)\ (\mathit{dur}\ m_2) \\
&\Rightarrow \mathit{dur}\ m_2
\end{aligned}
$$

Now we can continue the proof from above:

$$
\begin{aligned}
&\ldots \\
&\Rightarrow \textbf{let } d_1 = \mathit{dur}\ m_1 \\
&\qquad\quad d_2 = \mathit{dur}\ m_2 \\
&\quad\ \textbf{in if } d_1 > d_2 \textbf{ then } \mathit{dur}\ m_1 \\
&\qquad\qquad\qquad\quad \textbf{else } \mathit{dur}\ m_2 \\
&\Rightarrow \mathit{max}\ (\mathit{dur}\ m_1)\ (\mathit{dur}\ m_2) \\
&\Rightarrow \mathit{dur}\ (m_1 :=: m_2)
\end{aligned}
$$

The final inductive step involves the *Modify* constructor, but recall that *dur* treats a *Tempo* modification specially, and thus we treat it specially as well:

$$
\begin{aligned}
&\mathit{dur}\ (\mathit{revM}\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ m)) \\
&\Rightarrow \mathit{dur}\ (\mathit{Modify}\ (\mathit{Tempo}\ r)\ (\mathit{revM}\ m)) \\
&\Rightarrow \mathit{dur}\ (\mathit{revM}\ m) / r \\
&\Rightarrow \mathit{dur}\ m / r
\end{aligned}
$$

$$\Rightarrow dur\ (Modify\ (Tempo\ r)\ m)$$

Finally, we consider the case that $c \neq Tempo\ r$:

$$\begin{array}{l} dur\ (revM\ (Modify\ c\ m)) \\ \Rightarrow dur\ (Modify\ c\ (revM\ m)) \\ \Rightarrow Modify\ c\ (dur\ (revM\ m)) \\ \Rightarrow Modify\ c\ (dur\ m) \\ \Rightarrow dur\ (Modify\ c\ m) \end{array}$$

And we are done.

Exercise 10.4 Recall Exercises 3.11 and 3.12. Prove that, if $p_2 \geqslant p_1$:

$$chrom\ p_1\ p_2 = mkScale\ p_1\ (take\ (absPitch\ p_2 - absPitch\ p_1)\ (repeat\ 1))$$

using the lemma:

$$[m\mathinner{..}n] = scanl\ (+)\ m\ (take\ (n - m)\ (repeat\ 1))$$

Exercise 10.5 Prove the following facts involving dur:

$$\begin{array}{ll} dur\ (timesM\ n\ m) & = n * dur\ m \\ dur\ (takeM\ d\ m) & = d, \textbf{if}\ d \leqslant dur\ m \end{array}$$

Exercise 10.6 Prove the following facts involving $mMap$:

$$\begin{array}{ll} mMap\ id\ m & = m \\ mMap\ f\ (mMap\ g\ m) & = mMap\ (f \circ g)\ m \end{array}$$

Exercise 10.7 Prove that, for all $pmap$, c, and m:

$$perf\ pmap\ c\ m = (perform\ pmap\ c\ m, dur\ m)$$

where $perform$ is the function defined in Figure 8.1.

10.5.1 The Need for Musical Equivalence

In Chapter 1 we discussed the need for a notion of *musical equivalence*, noting that, for example, $m :+: rest\ 0$ "sounds the same" as m, even if the two *Music* values are not equal as Haskell values. That same issue can strike us here as we try to prove intuitively natural properties such as:

$$revM\ (revM\ m) = m$$

To see why this property cannot be proved without a notion of musical equivalence, note that:

$$
\begin{aligned}
&revM\ (revM\ (c\ 4\ en :=: d\ 4\ qn))\\
&\Longrightarrow revM\ ((rest\ en :+: c\ 4\ en) :=: d\ 4\ qn)\\
&\Longrightarrow (rest\ 0 :+: c\ 4\ en :+: rest\ en) :=: d\ 4\ qn
\end{aligned}
$$

Clearly the last line above is not equal, as a Haskell value, to $c\ 4\ en{:=:}d\ 4\ qn$. But somehow we need to show that these two values "sound the same" as musical values. In the next chapter we will formally develop the notion of musical equivalence, and with it be able to prove the validity of our intuitions regarding *revM*, as well as many other important musical properties.

10.6 [Advanced] Induction on Other Data Types

Proof by induction can be used to reason about many data types. For example, we can use it to reason about natural numbers.[2] Suppose we define an exponentiation function as follows:

$$
\begin{aligned}
&(\hat{\ })\ \ :: Integer \rightarrow Integer \rightarrow Integer\\
&x\,\hat{}\,0\ = 1\\
&x\,\hat{}\,n = x * x\,\hat{}\,(n-1)
\end{aligned}
$$

> **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 $n = 0$:

$$
\begin{aligned}
&x\,\hat{}\,(0+m)\\
&\Rightarrow x\,\hat{}\,m
\end{aligned}
$$

[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.

$\Rightarrow 1 * (x\,\hat{}\,m)$
$\Rightarrow x\,\hat{}\,0 * x\,\hat{}\,m$

Next we assume that the property is true for numbers less than or equal to n, and prove it for $n+1$:

$x\,\hat{}\,((n+1)+m)$
$\Rightarrow x * x\,\hat{}\,(n+m)$
$\Rightarrow x * (x\,\hat{}\,n * x\,\hat{}\,m)$
$\Rightarrow (x * x\,\hat{}\,n) * x\,\hat{}\,m$
$\Rightarrow x\,\hat{}\,(n+1) * x\,\hat{}\,m$

and we are done.

Or are we? What if, in the definition of $(\hat{})$, 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:

```
(^)                 :: Integer -> Integer -> Integer
x^0                 = 1
x^n | n < 0         = error "negative exponent"
    | otherwise     = x * x^(n-1)
```

If we consider non-terminating computations and ones that produce an error to both have the same value, namely $\perp$, then these two versions of $(\hat{})$ 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:

```
(^) :: Integer -> Integer -> Integer
x^n | 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.

10.6.1 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 (^!) and *ff* for our functions to distinguish them from the previous versions:

```
(^!) :: Integer -> Integer -> Integer
x ^! 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, (^) executes a number of steps proportional to n, whereas (^!) executes a number of steps proportional to the $\log_2$ of n. The Standard Prelude defines (^) similarly to the way in which (^!) 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 $x \mathbin{\hat{}} n = x \mathbin{\hat{}!} n$ for all x and n.

A quick look at the two definitions reveals that what we really need to prove is that $f\ x\ n = \mathit{ff}\ x\ n$, from which it follows immediately that $x \mathbin{\hat{}} n = x \mathbin{\hat{}!} n$. We do this by induction on n, beginning with the base case $n = 0$:

$$f\ x\ 0 \Rightarrow 1 \Rightarrow \mathit{ff}\ x\ 0$$

so the base case 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:

$$\begin{array}{l} f\ x\ (n+1) \\ \Rightarrow x * f\ x\ n \\ \Rightarrow x * \mathit{ff}\ x\ n \\ \Rightarrow \mathit{ff}\ x\ (n+1) \end{array}$$

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 does not 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:

$$\begin{array}{l} f\ x\ (n+1) \\ \Rightarrow f\ x\ ((n+1)\ \text{`}\mathit{quot}\text{`}\ 2 + (n+1)\ \text{`}\mathit{quot}\text{`}\ 2) \\ \Rightarrow f\ x\ ((n+1)\ \text{`}\mathit{quot}\text{`}\ 2) * f\ x\ ((n+1)\ \text{`}\mathit{quot}\text{`}\ 2) \end{array}$$

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 10.3: Proof that $f\ x\ n * f\ x\ n = f\ (x * x)\ n$.

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:

$$
\begin{array}{l}
f\ x\ ((n+1)\ \text{`}quot\text{`}\ 2) * f\ x\ ((n+1)\ \text{`}quot\text{`}\ 2) \\
\Rightarrow f\ (x * x)\ ((n+1)\ \text{`}quot\text{`}\ 2) \\
\Rightarrow ff\ (x * x)\ ((n+1)\ \text{`}quot\text{`}\ 2) \\
\Rightarrow ff\ x\ (n+1)
\end{array}
$$

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 10.3.

Aside from improving efficiency, one of the pleasant outcomes of proving that (^) and (^!) 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 should not 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 10.8 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 10.9 Consider this definition of the *factorial* function:[3]

$$
\begin{array}{l}
\mathit{fac}_1 \quad :: \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{fac}_1\ 0 = 1 \\
\mathit{fac}_1\ n = n * \mathit{fac}_1\ (n-1)
\end{array}
$$

and this alternative definition that uses an "accumulator:"

$$
\begin{array}{l}
\mathit{fac}_2 \quad :: \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{fac}_2\ n = \mathit{fac}'\ n\ 1 \\
\quad \textbf{where}\ \mathit{fac}'\ 0\ \mathit{acc} = \mathit{acc} \\
\qquad\qquad\ \mathit{fac}'\ n\ \mathit{acc} = \mathit{fac}'\ (n-1)\ (n * \mathit{acc})
\end{array}
$$

Prove that $\mathit{fac}_1 = \mathit{fac}_2$.

---

[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 11

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* [Hud04]. With this algebra we can reason about, transform, and optimize computer music programs in a meaning preserving way.

11.1 Musical Equivalance

Suppose we have two values $m_1 :: \mathit{Music\ Pitch}$ and $m_2 :: \mathit{Music\ Pitch}$, 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:

$$
\begin{aligned}
&m_1 = c\ 4\ \mathit{en} :\!+\!:\ d\ 4\ \mathit{qn} \\
&m_2 = \mathit{revM}\ (\mathit{revM}\ m_1)
\end{aligned}
$$

Then $m_1 == m_2$ is *True*.

Unfortunately, as we saw in the last chapter, if we reverse a parallel composition, things do not work out as well. For example:

$$
\begin{aligned}
&\mathit{revM}\ (\mathit{revM}\ (c\ 4\ \mathit{en} :\!=\!:\ d\ 4\ \mathit{qn})) \\
&\ \Rightarrow (\mathit{rest}\ 0 :\!+\!:\ c\ 4\ \mathit{en} :\!+\!:\ \mathit{rest}\ \mathit{en}) :\!=\!:\ d\ 4\ \mathit{qn}
\end{aligned}
$$

In addition, as we discussed briefly in Chapter 1, there are musical properties for which standard Haskell equivalence is insufficient to capture. For example, we would expect the following two musical values to *sound* the same, regardless of the actual values of m_1, m_2, and m_3:

$$(m_1 :+: m_2) :+: m_3$$
$$m_1 :+: (m_2 :+: m_3)$$

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 m_1, m_2, and m_3 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 m_1 and m_2 are *equivalent*, written $m_1 \equiv m_2$, if and only if:

$$(\forall pm, c) \;\; perf\ pm\ c\ m_1 = perf\ pm\ c\ m_2$$

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 8 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 11.1.1 For all pm, c, and m:

$$perf\ pm\ c\ m = (perform\ pm\ c\ m, dur\ m * cDur\ c)$$

where *perform* is the function defined in Figure 8.1.

To see the importance of including duration in the definition of equivalence, we first note that if two musical values are equivalent, we should

[1] If $m_1 = m_1 :+: m_2$ and $m_3 = m_2 :+: m_3$ then the two expressions are equal, but these are infinite values that cannot be reversed or even performed.

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, m_1 :+: *rest* 1 :+: m_2 is equivalent to m_1 :+: *rest* 2 :+: m_2, which is surely not what we want.[2]

Note that we could have defined *perf* as above, i.e. in terms of *perform* and *dur*, but as mentioned in Section 8.1 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 $m_1 \equiv m_2$ we need to prove:

$$\mathit{perf}\ \mathit{pm}\ c\ m_1 = \mathit{perf}\ \mathit{pm}\ c\ m_2$$

Instead of doing this directly using the definition of *perf*, we could instead prove both of the following:

$$\mathit{perform}\ \mathit{pm}\ c\ m_1 = \mathit{perform}\ \mathit{pm}\ c\ m_2$$
$$\mathit{dur}\ m_1 = \mathit{dur}\ m_2$$

11.1.1 Literal Player

The only problem with this strategy for defining musical equivalence is that the notion of a *player* can create situations where certain properties that we would like to hold, in fact do not. After all, a player may interpret a note or phrase in whatever way it (or he or she) may desire. For example, it seems that this property should hold:

$$\mathit{tempo}\ 1\ m \equiv m$$

However, a certain (rather perverse) player might interpret anything tagged with a *Tempo* modifier as an empty performance—in which case the above property will fail! To solve this problem, we assume that...

11.2 Some Simple Axioms

Let's look at a few simple axioms, and see how we can prove each of them using the proof techniques that we have developed so far.

(Note: In the remainder of this chapter we will use the functions *tempo* r and *trans* p to represent their unfolded versions, *Modify* (*Tempo* r) and

[2] A more striking example of this point is John Cage's composition *4'33"*, which consists basically of four minutes and thirty-three seconds of silence [].

Modify (*Transpose* t), respectively. In the proofs we will not bother with the intermediate steps of unfolding these functions.)

Here is the first axiom that we will consider:

Axiom 11.2.1 For any r_1, r_2, and m:

$$tempo\ r_1\ (tempo\ r_2\ m) \equiv tempo\ (r_1 * r_2)\ m$$

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:

$$\begin{array}{l}
\textbf{let}\ dt = cDur\ c \\
perform\ pm\ c\ (tempo\ r_1\ (tempo\ r_2\ m)) \\
\Rightarrow \{unfold\ perform\} \\
perform\ pm\ (c\ \{cDur = dt/r_1\})\ (tempo\ r_2\ m) \\
\Rightarrow \{unfold\ perform\} \\
perform\ pm\ (c\ \{cDur = (dt/r_1)/r_2\})\ m \\
\Rightarrow \{arithmetic\} \\
perform\ pm\ (c\ \{cDur = dt/(r_1 * r_2)\})\ m \\
\Rightarrow \{fold\ perform\} \\
perform\ pm\ c\ (tempo\ (r_1 * r_2)\ m)
\end{array}$$

$$\begin{array}{l}
dur\ (tempo\ r_1\ (tempo\ r_2\ m)) \\
\Rightarrow \{unfold\ dur\} \\
dur\ (tempo\ r_2\ m)/r_1 \\
\Rightarrow \{unfold\ dur\} \\
(dur\ m/r_2)/r_1 \\
\Rightarrow \{arithmetic\} \\
dur\ m/(r_1 * r_2) \\
\Rightarrow \{fold\ dur\} \\
dur\ (tempo\ (r_1 * r_2)\ m)
\end{array}$$

Here is another useful axiom and its proof:

Axiom 11.2.2 For any r, m_1, and m_2:

$$tempo\ r\ (m_1 :+: m_2) \equiv tempo\ r\ m_1 :+: tempo\ r\ m_2$$

In other words, *tempo scaling distributes over sequential composition.*

Proof:

$$
\begin{array}{l}
\textbf{let } t = cTime\ c; dt = cDur\ c \\
\quad t_1 = t + dur\ m_1 * (dt/r) \\
\quad t_2 = t + (dur\ m_1/r) * dt \\
\quad t3 = t + dur\ (tempo\ r\ m_1) * dt
\end{array}
$$

$$
\begin{array}{l}
perform\ pm\ c\ (tempo\ r\ (m_1 :+: m_2)) \\
\Rightarrow \{unfold\ perform\} \\
perform\ pm\ (c\ \{cDur = dt/r\})\ (m_1 :+: m_2) \\
\Rightarrow \{unfold\ perform\} \\
perform\ pm\ (c\ \{cDur = dt/r\})\ m_1 \\
\quad +\!\!+\ perform\ pm\ (c\ \{cTime = t_1, cDur = dt/r\})\ m_2 \\
\Rightarrow \{fold\ perform\} \\
perform\ pm\ c\ (tempo\ r\ m_1) \\
\quad +\!\!+\ perform\ pm\ (c\ \{cTime = t_1\})\ (tempo\ r\ m_2) \\
\Rightarrow \{arithmetic\} \\
perform\ pm\ c\ (tempo\ r\ m_1) \\
\quad +\!\!+\ perform\ pm\ (c\ \{cTime = t_2\})\ (tempo\ r\ m_2) \\
\Rightarrow \{fold\ dur\} \\
perform\ pm\ c\ (tempo\ r\ m_1) \\
\quad +\!\!+\ perform\ pm\ (c\ \{cTime = t3\})\ (tempo\ r\ m_2) \\
\Rightarrow \{fold\ perform\} \\
perform\ pm\ c\ (tempo\ r\ m_1 :+: tempo\ r\ m_2)
\end{array}
$$

$$
\begin{array}{l}
dur\ (tempo\ r\ (m_1 :+: m_2)) \\
\Rightarrow dur\ (m_1 :+: m_2)/r \\
\Rightarrow (dur\ m_1 + dur\ m_2)/r \\
\Rightarrow dur\ m_1/r + dur\ m_2/r \\
\Rightarrow dur\ (tempo\ r\ m_1) + dur\ (tempo\ r\ m_2) \\
\Rightarrow dur\ (tempo\ r\ m_1 :+: tempo\ r\ m_2)
\end{array}
$$

An even simpler axiom is given by:

Axiom 11.2.3 For any m, $tempo\ 1\ m \equiv m$.

In other words, *unit tempo scaling is the identity function for type Music.*

Proof:

$$
\textbf{let } dt = cDur\ c
$$

$$
\begin{array}{l}
perform\ pm\ c\ (tempo\ 1)\ m) \\
\Rightarrow \{unfold\ perform\} \\
perform\ pm\ (c\ \{cDur = dt/1\})\ m \\
\Rightarrow \{arithmetic\}
\end{array}
$$

$$perform\ pm\ c\ m$$

$$\begin{aligned}
& dur\ (tempo\ 1)\ m) \\
& \Rightarrow dur\ m/1 \\
& \Rightarrow dur\ m
\end{aligned}$$

Note that the above three proofs, being used to establish axioms, all involve the definitions of *perform* and *dur*. 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 11.2.1 For any r, m_1, and m_2:

$$tempo\ r\ m_1 :+: m_2 \equiv tempo\ r\ (m_1 :+: tempo\ (1/r)\ m_2)$$

Proof:

$$\begin{aligned}
& tempo\ r\ m_1 :+: m_2 \\
& \Rightarrow \{\,Axiom\ 3\} \\
& tempo\ r\ m_1 :+: tempo\ 1\ m_2 \\
& \Rightarrow \{\,arithmetic\} \\
& tempo\ r\ m_1 :+: tempo\ (r * (1/r))\ m_2 \\
& \Rightarrow \{\,Axiom\ 1\} \\
& tempo\ r\ m_1 :+: tempo\ r\ (tempo\ (1/r)\ m_2) \\
& \Rightarrow \{\,Axiom\ 2\} \\
& tempo\ r\ (m_1 :+: tempo\ (1/r)\ m_2)
\end{aligned}$$

11.3 The Fundamental 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 11.3.1 *Tempo* is *multiplicative* and *Transpose* is *additive*. That is, for any r_1, r_2, p_1, p_2, and m:

$$\begin{aligned}
tempo\ r_1\ (tempo\ r_2\ m) &\equiv tempo\ (r_1 * r_2)\ m \\
trans\ p_1\ (trans\ p_2\ m) &\equiv trans\ (p_1 + p_2)\ m
\end{aligned}$$

Axiom 11.3.2 Function composition is *commutative* with respect to both tempo scaling and transposition. That is, for any r_1, r_2, p_1 and p_2:

$$\begin{aligned}
&\mathit{tempo}\ r_1 \circ \mathit{tempo}\ r_2 \equiv \mathit{tempo}\ r_2 \circ \mathit{tempo}\ r_1\\
&\mathit{trans}\ p_1 \circ \mathit{trans}\ p_2 \ \ \equiv \mathit{trans}\ p_2 \circ \mathit{trans}\ p_1\\
&\mathit{tempo}\ r_1 \circ \mathit{trans}\ p_1 \ \equiv \mathit{trans}\ p_1 \circ \mathit{tempo}\ r_1
\end{aligned}$$

Axiom 11.3.3 Tempo scaling and transposition are *distributive* over both sequential and parallel composition. That is, for any r, p, m_1, and m_2:

$$\begin{aligned}
&\mathit{tempo}\ r\ (m_1 :+: m_2) \equiv \mathit{tempo}\ r\ m_1 :+: \mathit{tempo}\ r\ m_2\\
&\mathit{tempo}\ r\ (m_1 :=: m_2) \equiv \mathit{tempo}\ r\ m_1 :=: \mathit{tempo}\ r\ m_2\\
&\mathit{trans}\ p\ (m_1 :+: m_2) \ \equiv \mathit{trans}\ p\ m_1 :+: \mathit{trans}\ p\ m_2\\
&\mathit{trans}\ p\ (m_1 :=: m_2) \ \equiv \mathit{trans}\ p\ m_1 :=: \mathit{trans}\ p\ m_2
\end{aligned}$$

Axiom 11.3.4 Sequential and parallel composition are *associative*. That is, for any m_0, m_1, and m_2:

$$\begin{aligned}
&m_0 :+: (m_1 :+: m_2) \ \equiv (m_0 :+: m_1) :+: m_2\\
&m_0 :=: (m_1 :=: m_2) \equiv (m_0 :=: m_1) :=: m_2
\end{aligned}$$

Axiom 11.3.5 Parallel composition is *commutative*. That is, for any m_0 and m_1:

$$m_0 :=: m_1 \equiv m_1 :=: m_0$$

Axiom 11.3.6 *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:

$$\begin{aligned}
&\mathit{tempo}\ r\ (\mathit{rest}\ 0) \equiv \mathit{rest}\ 0\\
&\mathit{trans}\ p\ (\mathit{rest}\ 0) \ \equiv \mathit{rest}\ 0\\
&m :+: \mathit{rest}\ 0 \qquad \equiv m \equiv \mathit{rest}\ 0 :+: m\\
&m :=: \mathit{rest}\ 0 \qquad \equiv m \equiv \mathit{rest}\ 0 :=: m
\end{aligned}$$

Axiom 11.3.7 A rest can be used to "pad" a parallel composition. That is, for any m_1, m_2, such that $\mathit{diff} = \mathit{dur}\ m_1 > \mathit{dur}\ m_2 \geqslant 0$, and any $d \leqslant \mathit{diff}$:

$$m_1 :=: m_2 \equiv m_1 :=: (m_2 :+: \mathit{rest}\ d)$$

Axiom 11.3.8 There is a duality between (:+:) and (:=:), namely that, for any m_0, m_1, m_2, and m_3 such that $\mathit{dur}\ m_0 = \mathit{dur}\ m_2$:

$$(m_0 :+: m_1) :=: (m_2 :+: m_3) \equiv (m_0 :=: m_2) :+: (m_1 :=: m_3)$$

Exercise 11.1 Prove Lemma 11.1.1.

Exercise 11.2 Establish the validity of each of the above axioms.

Exercise 11.3 Recall the polyphonic and contrapuntal melodies mel_1 and mel_2 from Chapter 1. Prove that $mel_1 \equiv mel_2$.

11.4 An Algebraic Semantics

Discuss formal semantics. Denotational, operational (relate to "proof by calculation"), and algebraic.

Soundness and Completeness.

[Hud04]

11.5 Other Musical Properties

Aside from the axioms discussed so far, there are many other properties of *Music* values and its various operators, just as we saw in Chapter 10 for lists. For example, this property of *map* taken from Figure 10.1:

$$map\ (f \circ g) = map\ f \circ map\ g$$

suggests and analogous property for *mMap*:

$$map\ (f \circ g) = map\ f \circ map\ g$$

Not all of the properties in Figures 10.1 and 10.2 have analogous musical renditions, and there are also others that are special only to *Music* values. Figure 11.1 summarizes the most important of these properties, including the one above. Note that some of the properties are expressed as strict equality—that is, the left-hand and right-hand sides are equivalent as Haskell values. But others are expressed using musical equivalence—that is, using $(\equiv)$. We leave the proofs of all these properties as an exercise.

Exercise 11.4 Prove that $timesM\ \ a\ \ m$:+: $timesM\ \ b\ \ m \equiv timesM\ \ (a + b)\ m$.

Exercise 11.5 Prove as many of the axioms from Figure 11.1 as you can.

Properties of $mMap$**:**

$$
\begin{aligned}
mMap\ (\lambda x \to x) &= \lambda x \to x \\
mMap\ (f \circ g) &= mMap\ f \circ mMap\ g \\
mMap\ f \circ dropM\ \ d &= dropM\ \ d \circ mMap\ f \\
mMap\ f \circ takeM\ \ d &= takeM\ \ d \circ mMap\ f
\end{aligned}
$$

Properties of $takeM$ **and** $dropM$**:**

For all non-negative d_1 and d_2:

$$
\begin{aligned}
takeM\ \ d_1 \circ takeM\ \ d_2 &= takeM\ (min\ d_1\ d_2) \\
dropM\ \ d_1 \circ dropM\ \ d_2 &= dropM\ (d_1 + d_2) \\
takeM\ \ d_1 \circ dropM\ \ d_2 &= dropM\ \ d_1 \circ takeM\ (d_1 + d_2)
\end{aligned}
$$

For all non-negative d_1 and d_2 such that $d_2 \geqslant d_1$:

$$dropM\ \ d_1 \circ takeM\ \ d_2 = takeM\ (d_2 - d_1) \circ dropM\ \ d_1$$

Properties of $revM$**:**

For all finite-duration m:

$$
\begin{aligned}
revM\ (revM\ \ m) &\equiv m \\
revM\ (takeM\ \ d\ \ m) &\equiv dropM\ (dur\ \ m - d)\ (revM\ \ m) \\
revM\ (dropM\ \ d\ \ m) &\equiv takeM\ (dur\ \ m - d)\ (revM\ \ m) \\
takeM\ \ d\ (revM\ \ m) &\equiv revM\ (dropM\ (dur\ \ m - d)\ m) \\
dropM\ \ d\ (revM\ \ m) &\equiv revM\ (takeM\ (dur\ \ m - d)\ m)
\end{aligned}
$$

Properties of dur**:**

$$
\begin{aligned}
dur\ (revM\ \ m) &= dur\ \ m \\
dur\ (takeM\ \ d\ \ m) &= min\ d\ (dur\ \ m) \\
dur\ (dropM\ \ d\ \ m) &= max\ 0\ (dur\ \ m - d)
\end{aligned}
$$

Figure 11.1: Useful Properties of Other Musical Functions

Chapter 12

Musical L-Systems and Generative Grammars

module *Euterpea.Examples.LSystems* **where**
import *Euterpea*
import *Data.List hiding* (*transpose*)
import *System.Random*

12.1 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 $X \to Y$. X and Y are sentences (or *sentential forms*) formed 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 non-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.

12.1.1 A Simple Implementation

A framework for simple, context-free, deterministic grammars can be designed in Haskell 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)
```

Details: *maybe* is a convenient function for conditionally giving a result based on the structure of a value of type *Maybe a*. It is defined in the Standard Prelude as:

$$
\begin{array}{l}
maybe \quad :: b \to (a \to b) \to Maybe\ a \to b \\
maybe\ \_\ f\ (Just\ x) = f\ x \\
maybe\ z\ \_\ Nothing = z
\end{array}
$$

$lookup :: Eq\ a \Rightarrow a \to [(a, b)] \to Maybe\ b$ is a convenient function for finding the value associated with a given key in an association list. For example:

$$
\begin{array}{l}
lookup\ \texttt{'b'}\ [(\texttt{'a'}, 0), (\texttt{'b'}, 1), (\texttt{'c'}, 2)] \Rightarrow Just\ 1 \\
lookup\ \texttt{'d'}\ [(\texttt{'a'}, 0), (\texttt{'b'}, 1), (\texttt{'c'}, 2)] \Rightarrow Nothing
\end{array}
$$

Note that we expand 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 *Data.List* 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"),('|',"|"),
   ('(',"("),    (')',")"),('/',"\\"),
   ('\\',"/")
  ]
```

Details: Recall that `'\\'` is how the backslash character is written in Haskell, because a single backslash is the "escape" character for writing special characters such as newline (`'\n'`), tab (`'\t'`), and so on. Since the backslash is used in this way, it also is a special character, and must be escaped using itself, i.e. `'\\'`.

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, *t* 10 *redAlgae* yields:

```
a
b|c
b|b|d
b|b|e\d
b|b|f/e\d
b|b|g\f/e\d
b|b|h(a)/g\f/e\d
b|b|h(b|c)\h(a)/g\f/e\d
b|b|h(b|b|d)/h(b|c)\h(a)/g\f/e\d
b|b|h(b|b|e\d)\h(b|b|d)/h(b|c)\h(a)/g\f/e\d
```

Exercise 12.1 Define a function $strToMusic :: AbsPitch \rightarrow Dur \rightarrow String \rightarrow Music\ Pitch$ that interprets the strings generated by *redAlgae* as music. Specifically, *strToMusic ap d str* interprets the string *str* in the following way:

1. Characters `'a'` through `'h'` are interpreted as notes, each with duration d and absolute pitches ap, $ap + 2$, $ap + 4$, $ap + 5$, $ap + 7$, $ap + 9$, $ap + 11$, and $ap + 12$, respectively (i.e. a major scale).
2. `'|'` is interpreted as a no-op.
3. `'/'` and `'\\'` are both interpreted as a rest of length d.
4. `'('` is interpreted as a transposition by 5 semitones (a perfect fourth).
5. `')'` is interpreted as a transposition by -5 semitones.

Exercise 12.2 Design a function $testDet :: Grammar\ a \rightarrow Bool$ 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.

12.1.2 A More General Implementation

The design given in the last section only captures deterministic context-free grammars, and the generator considers only parallel productions that are charactersitic of L-Systems.

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 = Double
```

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      = Double
```

The idea here 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 divided by 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)
```

Details: *groupBy* :: (*a* → *a* → *Bool*) → [*a*] → [[*a*]] is a *Data.List* library function that behaves as follows: *groupBy eqfn xs* returns a list of lists such that all elements in each sublist are "equal" in the sense defined by *eqfn*.

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 replacement function, a list of rules, 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)
```

A key aspect of the *generate* algorithm above is to compute the *probability density* of each successive rule, which is basically the sum of its probability plus the probabilities of all rules that precede it.

12.2 An L-System Grammar for Music

The previous section gave a generative framework 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 s_1 :+ s_2.
- A functional composition s_1 :. s_2.
- The symbol *Id*, which will eventually be 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)
```

The idea here is that sentences generated from this grammar are relative to a starting note, and thus the above constructions will be interpreted as functions that take that starting note as an argument. This will all become clear shortly, but first we need to define a replacement function for this grammar.

We will treat (:+) and (:.) as binary branches, and recursively traverse each of their arguments. We will 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 replacement term for it.

$$
\begin{array}{l}
\mathit{replFun} :: \mathit{Eq}\ a \Rightarrow \mathit{ReplFun}\ (\mathit{LSys}\ a)\\
\mathit{replFun}\ \mathit{rules}\ (s, \mathit{rands}) =\\
\quad \mathbf{case}\ s\ \mathbf{of}\\
\qquad a :\!+\ b \rightarrow \mathbf{let}\ (a', \mathit{rands}') \ = \mathit{replFun}\ \mathit{rules}\ (a, \mathit{rands})\\
\qquad\qquad\qquad\quad\ (b', \mathit{rands}'') = \mathit{replFun}\ \mathit{rules}\ (b, \mathit{rands}')\\
\qquad\qquad\quad\ \ \mathbf{in}\ (a' :\!+\ b', \mathit{rands}'')\\
\qquad a :\!.\ b \ \ \rightarrow \mathbf{let}\ (a', \mathit{rands}') \ = \mathit{replFun}\ \mathit{rules}\ (a, \mathit{rands})\\
\qquad\qquad\qquad\quad\ (b', \mathit{rands}'') = \mathit{replFun}\ \mathit{rules}\ (b, \mathit{rands}')\\
\qquad\qquad\quad\ \ \mathbf{in}\ (a' :\!.\ b', \mathit{rands}'')\\
\qquad \mathit{Id} \qquad \rightarrow (\mathit{Id}, \mathit{rands})\\
\qquad N\ x \quad\ \rightarrow (\mathit{getNewRHS}\ \mathit{rules}\ (N\ x)\ (\mathit{head}\ \mathit{rands}), \mathit{tail}\ \mathit{rands})
\end{array}
$$

getNewRHS is defined as:

$$
\begin{array}{l}
\mathit{getNewRHS} :: \mathit{Eq}\ a \Rightarrow [[(\mathit{Rule}\ a, \mathit{Prob})]] \rightarrow a \rightarrow \mathit{Rand} \rightarrow a\\
\mathit{getNewRHS}\ \mathit{rrs}\ \mathit{ls}\ \mathit{rand} =\\
\quad \mathbf{let}\ \mathit{loop}\ ((r, p) : \mathit{rs}) = \mathbf{if}\ \mathit{rand} \leqslant p\ \mathbf{then}\ \mathit{rhs}\ r\ \mathbf{else}\ \mathit{loop}\ \mathit{rs}\\
\qquad\quad \mathit{loop}\ [\,] \qquad\qquad\ = \mathit{error}\ \texttt{"getNewRHS anomaly"}\\
\quad \mathbf{in}\ \mathbf{case}\ (\mathit{find}\ (\lambda((r, p){:}\ \_) \rightarrow \mathit{lhs}\ r == \mathit{ls})\ \mathit{rrs})\ \mathbf{of}\\
\qquad \mathit{Just}\ \mathit{rs} \ \ \rightarrow \mathit{loop}\ \mathit{rs}\\
\qquad \mathit{Nothing} \rightarrow \mathit{error}\ \texttt{"No rule match"}
\end{array}
$$

> **Details:** $\mathit{find} :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow \mathit{Maybe}\ a$ is another *Data.List* function that returns the first element of a list that satisfies a predicate, or *Nothing* if there is no such element.

12.2.1 Examples

The final step is to interpret the resulting sentence (i.e. a value of type *LSys a*) as music. As mentioned earlier, 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:

$$
\begin{array}{l}
\textbf{type}\ \mathit{IR}\ a\ b = [(a, \mathit{Music}\ b \rightarrow \mathit{Music}\ b)] \quad \text{-- ``interpetation rules''} \\
\mathit{interpret} :: (\mathit{Eq}\ a) \Rightarrow \mathit{LSys}\ a \rightarrow \mathit{IR}\ a\ b \rightarrow \mathit{Music}\ b \rightarrow \mathit{Music}\ b \\
\mathit{interpret}\ (a :.\ b)\ \ r\ m = \mathit{interpret}\ a\ r\ (\mathit{interpret}\ b\ r\ m) \\
\mathit{interpret}\ (a :+\ b)\ r\ m = \mathit{interpret}\ a\ r\ m :+: \mathit{interpret}\ b\ r\ m \\
\mathit{interpret}\ \mathit{Id}\qquad\ \ r\ m = m \\
\mathit{interpret}\ (N\ x)\quad r\ m = \textbf{case}\ (\mathit{lookup}\ x\ r)\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ f\quad\ \rightarrow f\ m \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{error}\ \texttt{"No interpetation rule"}
\end{array}
$$

For example, we could define the following interpretation rules:

$$
\begin{array}{l}
\textbf{data}\ \mathit{LFun} = \mathit{Inc} \mid \mathit{Dec} \mid \mathit{Same} \\
\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show}) \\
\mathit{ir} :: \mathit{IR}\ \mathit{LFun}\ \mathit{Pitch} \\
\mathit{ir} = [(\mathit{Inc}, \mathit{transpose}\ 1), \\
\quad (\mathit{Dec}, \mathit{transpose}\ (-1)), \\
\quad (\mathit{Same}, \mathit{id})] \\
\mathit{inc}, \mathit{dec}, \mathit{same} :: \mathit{LSys}\ \mathit{LFun} \\
\mathit{inc}\quad = N\ \mathit{Inc} \\
\mathit{dec}\quad = N\ \mathit{Dec} \\
\mathit{same} = N\ \mathit{Same}
\end{array}
$$

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:

$$\mathit{sc} = \mathit{inc} :+ \mathit{dec}$$

Now let's define a bunch of rules as follows:

$$
\begin{array}{l}
\mathit{r1a} = \mathit{Rule}\ \mathit{inc}\ (\mathit{sc} :.\ \mathit{sc}) \\
\mathit{r1b} = \mathit{Rule}\ \mathit{inc}\ \mathit{sc} \\
\mathit{r2a} = \mathit{Rule}\ \mathit{dec}\ (\mathit{sc} :.\ \mathit{sc}) \\
\mathit{r2b} = \mathit{Rule}\ \mathit{dec}\ \mathit{sc} \\
\mathit{r3a} = \mathit{Rule}\ \mathit{same}\ \mathit{inc} \\
\mathit{r3b} = \mathit{Rule}\ \mathit{same}\ \mathit{dec} \\
\mathit{r3c} = \mathit{Rule}\ \mathit{same}\ \mathit{same}
\end{array}
$$

and the corresponding grammar:

$$\mathit{g1} = \mathit{Grammar}\ \mathit{same}\ (\mathit{Uni}\ [\mathit{r1b}, \mathit{r1a}, \mathit{r2b}, \mathit{r2a}, \mathit{r3a}, \mathit{r3b}])$$

Finally, we generate a sentence at some particular level, and interpret it as music:

$$
\begin{aligned}
t_1\ n = {} & \mathit{instrument}\ \mathit{Vibraphone}\ \$ \\
& \mathit{interpret}\ (\mathit{gen}\ \mathit{replFun}\ \mathit{g1}\ 42\ !!\ n)\ \mathit{ir}\ (c\ 5\ \mathit{tn})
\end{aligned}
$$

Try "*play* (t_1 3)" or "*play* (t_1 4)" to hear the result.

Exercise 12.3 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 12.4 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 13

Random Numbers, Probability Distributions, and Markov Chains

```
module Euterpea.Examples.RandomMusic where
import Euterpea
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*.

13.1 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:

$$\mathit{mkStdGen} :: \mathit{Int} \rightarrow \mathit{StdGen}$$

which takes an *Int* seed as argument, and returns a "standard generator" of type *StdGen*. For example, we can define:

$$\begin{array}{l} \mathit{sGen} :: \mathit{StdGen} \\ \mathit{sGen} = \mathit{mkStdGen}\ 42 \end{array}$$

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:

$$\begin{array}{l} \textbf{class}\ \mathit{RandomGen}\ g\ \textbf{where} \\ \quad \begin{array}{ll} \mathit{genRange} & :: g \rightarrow (\mathit{Int}, \mathit{Int}) \\ \mathit{next} & :: g \rightarrow (\mathit{Int}, g) \\ \mathit{split} & :: g \rightarrow (g, g) \end{array} \end{array}$$

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:

$$\begin{array}{l} \mathit{randInts} :: \mathit{StdGen} \rightarrow [\mathit{Int}] \\ \mathit{randInts}\ g = \textbf{let}\ (x, g') = \mathit{next}\ g \\ \qquad\qquad\qquad\ \textbf{in}\ x : \mathit{randInts}\ g' \end{array}$$

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:

$$\begin{array}{l} \textbf{class}\ \mathit{Random}\ a\ \textbf{where} \\ \quad \mathit{randomR} \quad :: \mathit{RandomGen}\ g \Rightarrow (a, a) \rightarrow g \rightarrow (a, g) \end{array}$$

$$\begin{array}{ll} random & :: RandomGen\ g \Rightarrow g \rightarrow (a, g) \\ randomRs & :: RandomGen\ g \Rightarrow (a, a) \rightarrow g \rightarrow [a] \\ randoms & :: RandomGen\ g \Rightarrow g \rightarrow [a] \\ randomRIO & :: (a, a) \rightarrow IO\ a \\ randomIO & :: IO\ a \end{array}$$

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 serve 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:

$$
\begin{aligned}
&randIO' :: IO\ ()\\
&randIO' = \textbf{do}\ r_1 \leftarrow randomRIO\ (0,1) :: IO\ Float\\
&\qquad\qquad\qquad r_2 \leftarrow randomRIO\ (0,1) :: IO\ Float\\
&\qquad\qquad\qquad print\ (r_1 == r_2)
\end{aligned}
$$

will almost always return *False*, because the chance of two randomly generated floating point numbers being the same is exceedingly small. (The type signature is needed to ensure that the value generated has an unambigous type.)

Details: $print :: Show\ a \Rightarrow a \rightarrow IO\ ()$ converts any showable value into a string, and displays the result in the standard output area.

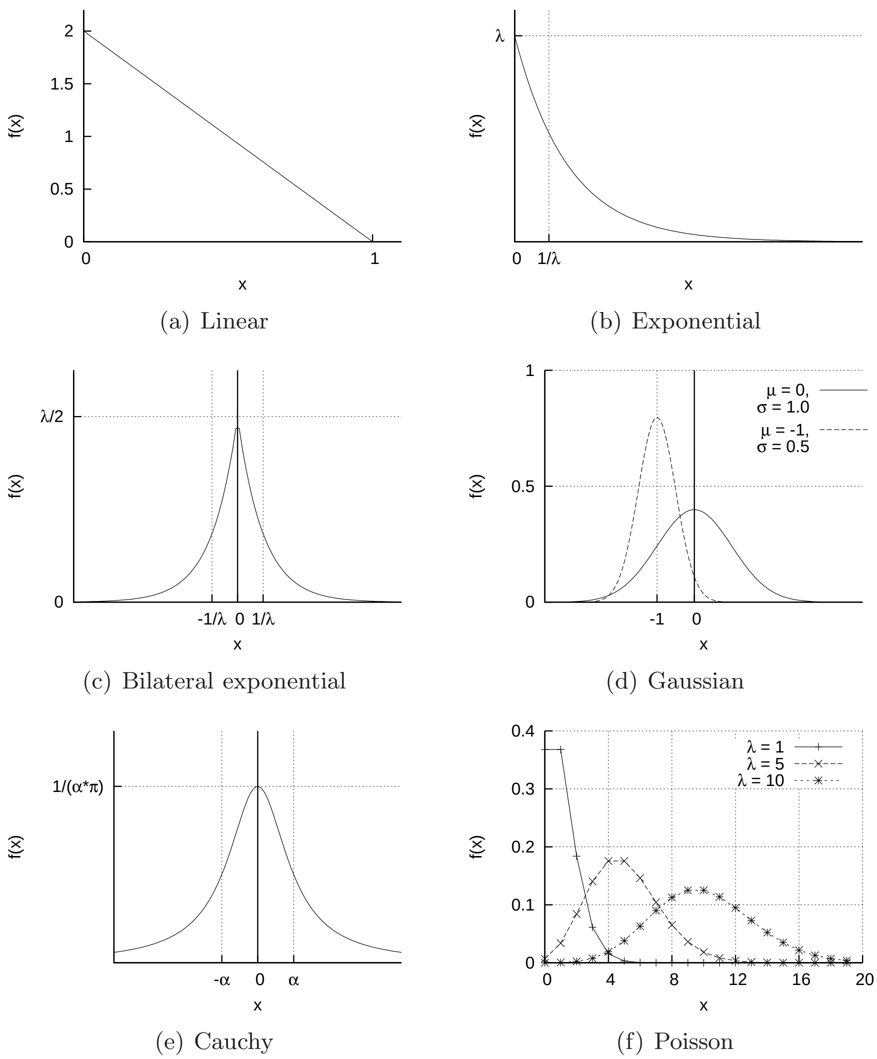
13.2 Probability Distributions

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 13.1 shows the probability density functions for each of othem.

Here is a list and brief description of each random number generator:



(a) Linear

(b) Exponential

(c) Bilateral exponential

(d) Gaussian

(e) Cauchy

(f) Poisson

Figure 13.1: Various Probability Density Functions

linear Generates a *linearly* distributed random variable between 0 and 1. 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:

$$\begin{aligned} &\mathit{exponential} :: (\mathit{RandomGen}\ g, \mathit{Floating}\ a, \mathit{Random}\ a) \Rightarrow \\ &\qquad a \to g \to (a, g) \end{aligned}$$

The first argument is the parameter λ.

bilateral exponential Generates a random number with a *bilateral exponential* distribution. It is 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:

$$\begin{aligned} &\mathit{gaussian} :: (\mathit{Floating}\ a, \mathit{Random}\ a, \mathit{RandomGen}\ g) \Rightarrow \\ &\qquad a \to a \to g \to (a, g) \end{aligned}$$

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:

$$\begin{aligned} &\mathit{cauchy} :: (\mathit{Floating}\ a, \mathit{Random}\ a, \mathit{RandomGen}\ g) \Rightarrow \\ &\qquad a \to g \to (a, g) \end{aligned}$$

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:

$$\begin{aligned} &\mathit{poisson} :: (\mathit{Num}\ t, \mathit{Ord}\ a, \mathit{Floating}\ a, \mathit{Random}\ a \\ &\qquad\quad \mathit{RandomGen}\ g) \Rightarrow \\ &\qquad a \to g \to (t, g) \end{aligned}$$

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.

$$\begin{array}{l} \mathit{frequency} :: (\mathit{Floating}\ w, \mathit{Ord}\ w, \mathit{Random}\ w, \mathit{RandomGen}\ g) \Rightarrow \\ \qquad [(w, a)] \rightarrow g \rightarrow (a, g) \end{array}$$

13.2.1 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:

$$... \rightarrow g \rightarrow (a, g)$$

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:

$$\begin{array}{l} \mathit{rands} \quad :: (\mathit{RandomGen}\ g, \mathit{Random}\ a) \Rightarrow \\ \qquad (g \rightarrow (a, g)) \rightarrow g \rightarrow [a] \\ \mathit{rands}\ f\ g = x : \mathit{rands}\ f\ g'\ \mathbf{where}\ (x, g') = f\ g \end{array}$$

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:

$$\begin{array}{l} \mathit{toAbsP}_1 \quad :: \mathit{Float} \rightarrow \mathit{AbsPitch} \\ \mathit{toAbsP}_1\ x = \mathit{round}\ (40 * x + 30) \end{array}$$

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:

$$\begin{array}{l} \mathit{mkNote}_1 :: \mathit{AbsPitch} \rightarrow \mathit{Music}\ \mathit{Pitch} \\ \mathit{mkNote}_1 = \mathit{note}\ \mathit{tn} \circ \mathit{pitch} \\ \mathit{mkLine}_1 \qquad :: [\mathit{AbsPitch}] \rightarrow \mathit{Music}\ \mathit{Pitch} \\ \mathit{mkLine}_1\ \mathit{rands} = \mathit{line}\ (\mathit{take}\ 32\ (\mathit{map}\ \mathit{mkNote}_1\ \mathit{rands})) \end{array}$$

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 13.1 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 13.2 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 $toAbsP_2$ does something reasonable to interpret a floating-point number as an interval, and $mkLine_2$ uses *scanl* to generate a "running sum" that represents the melody line.

13.3 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.

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 13.1 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. The

	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 13.1: Second-Order Markov Chain

resulting stochastic system is called a *Markov Chain*.

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]

13.3.1 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
```

$$
\begin{array}{l}
ps_0, ps_1, ps_2 :: [\mathit{Pitch}] \\
ps_0 = [(C,4),(D,4),(E,4)] \\
ps_1 = [(C,4),(D,4),(E,4),(F,4),(G,4),(A,4),(B,4)] \\
ps_2 = [(C,4),(E,4),(G,4),(E,4),(F,4),(A,4),(G,4),(E,4), \\
\qquad\quad (C,4),(E,4),(G,4),(E,4),(F,4),(D,4),(C,4)]
\end{array}
$$

$$
\begin{array}{l}
\quad \text{-- functions to package up } \mathit{run} \text{ and } \mathit{runMulti} \\
\mathit{mc} \quad \mathit{ps} \quad n = \mathit{mkLine}_3\ (M.\mathit{run}\ n\ \mathit{ps}\ 0\ (\mathit{mkStdGen}\ 42)) \\
\mathit{mcm}\ \mathit{pss}\ n = \mathit{mkLine}_3\ (\mathit{concat}\ (M.\mathit{runMulti}\ n\ \mathit{pss}\ 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{mkStdGen}\ 42)))
\end{array}
$$

$$
\begin{array}{l}
\quad \text{-- music-making functions} \\
\mathit{mkNote}_3 \quad :: \mathit{Pitch} \rightarrow \mathit{Music}\ \mathit{Pitch} \\
\mathit{mkNote}_3 \quad = \mathit{note}\ \mathit{tn} \\
\mathit{mkLine}_3 \quad :: [\mathit{Pitch}] \rightarrow \mathit{Music}\ \mathit{Pitch} \\
\mathit{mkLine}_3\ \mathit{ps} = \mathit{line}\ (\mathit{take}\ 64\ (\mathit{map}\ \mathit{mkNote}_3\ \mathit{ps}))
\end{array}
$$

Here are some things to try with the above definitions:

- $mc\ ps_0\ 0$ will generate a completely random sequence, since it is a "zeroth-order" Markov Chain that does not look at any previous output.
- $mc\ ps_0\ 1$ 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.
- $mc\ ps_1\ 1$ also generates a result that sounds just like its training data.
- $mc\ ps_2\ 1$, 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.
- $mcm\ [ps_0, ps_2]\ 1$ and $mcm\ [ps_1, ps_2]\ 1$ generate perhaps the most interesting results yet, in which you can hear aspects of both the ascending melodic nature of ps_0 and ps_1, and the harmonic structure of ps_2.
- $mcm\ [ps_1, reverse\ \ ps_1]\ 1$ has, not suprisingly, both ascending and descending lines in it, as reflected in the training data.

Exercise 13.3 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 14

From Performance to Midi

```haskell
module Euterpea.IO.MIDI.ToMidi (toMidi, UserPatchMap, defST,
  defUpm, testMidi, testMidiA,
  test, testA, writeMidi, writeMidiA,
  play, playM, playA,
  makeMidi, mToMF, gmUpm, gmTest) where
import Euterpea.Music.Note.Music
import Euterpea.Music.Note.MoreMusic
import Euterpea.Music.Note.Performance
import Euterpea.IO.MIDI.GeneralMidi
import Euterpea.IO.MIDI.MidiIO
import Euterpea.IO.MIDI.ExportMidiFile
import Sound.PortMidi
import Data.List (partition)
import Data.Char (toLower, toUpper)
import Codec.Midi
```

writeMidi :: (Performable a) =¿ FilePath -¿ Music a -¿ IO () writeMidi fn = exportMidiFile fn . testMidi

writeMidiA :: (Performable a) =¿ FilePath -¿ PMap Note1 -¿ Context Note1 -¿ Music a -¿ IO () writeMidiA fn pm con m = exportMidiFile fn (testMidiA pm con m)

Midi is shorthand for "Musical Instrument Digital Interface," and is a standard protocol for controlling electronic musical instruments [Ass13a, Ass13b]. This chapter describes how to convert an abstract *performance* as defined in Chapter 8 into a *standard Midi file* that can be played on any modern PC with a standard sound card.

14.1 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. 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.

14.1.1 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 14.1.

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,

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 14.1: General Midi Instrument Families

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:

$$
\begin{array}{l}
\mathit{toGM} :: \mathit{InstrumentName} \rightarrow \mathit{ProgNum} \\
\mathit{toGM}\ \mathit{Percussion} = 0 \\
\mathit{toGM}\ (\mathit{Custom}\ \mathit{name}) = 0 \\
\mathit{toGM}\ \mathbf{in} = \mathit{fromEnum}\ \mathbf{in} \\
\\
\mathbf{type}\ \mathit{ProgNum} = \mathit{Int}
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{fromGM} :: \mathit{ProgNum} \rightarrow \mathit{InstrumentName} \\
\mathit{fromGM}\ \mathit{pn} \mid \mathit{pn} \geqslant 0 \wedge \mathit{pn} \leqslant 127 = \mathit{fromEnum}\ \mathit{pn} \\
\mathit{fromGM}\ \mathit{pn} = \mathit{error}\ (\texttt{"fromGM: "} +\!\!+\ \mathit{show}\ \mathit{pn} +\!\!+ \\
\quad \texttt{" is not a valid General Midi program number"})
\end{array}
$$

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.

14.1.2 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:

$$\textbf{type}\ \mathit{UserPatchMap} = [(\mathit{InstrumentName}, \mathit{Channel})]$$

$$\textbf{type}\ \mathit{Channel} = \mathit{Int}$$

The only thing odd about Midi Channels is that General Midi specifies that Channel 10 (9 in Euterpea's 0-based numbering) is dedicated to *percussion* (which is different from the "percussive instruments" described in Table 14.1). 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 6 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*). Euterpea'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 "MakeGMMap: Too many instruments."
    mkGMMap n (Percussion : ins) =
      (Percussion, 9) : mkGMMap n ins
    mkGMMap n (i : ins) =
      (i, chanList !! n) : mkGMMap (n + 1) ins
```

$$chanList = [0\,..\,8] +\!\!+ [10\,..\,15] \quad \text{-- channel 9 is for percussion}$$

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:

$$
\begin{aligned}
&upmLookup :: UserPatchMap \rightarrow InstrumentName \\
&\qquad\qquad\qquad\qquad\qquad\quad \rightarrow (Channel, ProgNum) \\
&upmLookup\ upm\ iName = (chan, toGM\ iName) \\
&\quad \mathbf{where}\ chan = maybe\ (error\ (\texttt{"instrument "} +\!\!+ show\ iName +\!\!+ \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{" not in patch map"})) \\
&\qquad\qquad\qquad\qquad\quad id\ (lookup\ iName\ upm)
\end{aligned}
$$

14.1.3 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 Euterpea 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 Euterpea's functionality with direct Midi capability.

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 14.1 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] 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.

```
-- 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 (\acc t -> (t, t - acc)) 0 ts
```

Figure 14.1: Partial Definition of the *Midi* Data Type

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").

4. There are two kinds of *Message*s: *channel messages* and *meta messages*. Figure 14.1 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.

14.2 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 split    = splitByInst pf
      insts    = map fst split
      rightMap = if (allValid upm insts) then upm
                 else (makeGMMap insts)
  in Midi (if length split == 1 then SingleTrack
                                else MultiTrack)
          (TicksPerBeat division)
          (map (fromAbsTime ∘ performToMEvs rightMap) split)
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 pf_1 those events that use the same instrument as the first event in the performance. The rest of the events are collected into pf_2, 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 (chan, progNum) = upmLookup upm inm
      setupInst       = (0, ProgramChange chan progNum)
      setTempo        = (0, TempoChange defST)
      loop []     = []
      loop (e : es) = let (mev1, mev2) = mkMEvents chan e
                      in mev1 : insertMEvent mev2 (loop es)
  in setupInst : setTempo : loop pf
```

A source of incompatibilty between Euterpea and Midi is that Euterpea 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 Euterpea *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 * 2.0 * 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.

$$
\begin{array}{l}
\mathit{insertMEvent} :: \mathit{MEvent} \to [\,\mathit{MEvent}\,] \to [\,\mathit{MEvent}\,] \\
\mathit{insertMEvent}\ \mathit{mev1}\ [\,] = [\,\mathit{mev1}\,] \\
\mathit{insertMEvent}\ \mathit{mev1}@(t_1, \_)\ \mathit{mevs}@(\mathit{mev2}@(t_2, \_) : \mathit{mevs}') = \\
\quad \textbf{if}\ t_1 \leqslant t_2\ \textbf{then}\ \mathit{mev1} : \mathit{mevs} \\
\qquad \textbf{else}\ \mathit{mev2} : \mathit{insertMEvent}\ \mathit{mev1}\ \mathit{mevs}'
\end{array}
$$

14.3 Putting It All Together

Chapter 15

Basic Input/Output

So far the only input/output (IO) that we have seen in Euterpea is the use of the *play* function to generate the MIDI output corresponding to a *Music* value. But we have 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 Chapter 17 we will describe an elegant way to do IO involving a "musical user interface," or *MUI*.

15.1 IO in Haskell

The Haskell Report defines the result of a program to be the value of the variable *main* in the module *Main*. This is a mere technicality, however, only having relevance when you compile a program as a stand-alone executable (see the GHC documentation for a discussion of how to do that).

The way most people run Haskell programs, especially during program development, is through the GHCi command prompt. 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. GHC does this by insisting that the type of the value be an instance of the *Show* class—in which case it "shows" the result by converting it to a string using the *show* function (recall the discussion in Section 7.1). 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*.

Some commands return a value for subsequent use by the program: a character from the keyboard, 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."

15.2 do Syntax

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 $String \to IO$ (). 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 one 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:

$\mathbf{do}\ s \leftarrow \mathit{getLine}$
$\quad \mathit{writeFile}$ `"testFile.txt"` s

> **Details:** Note the syntax for binding s to the result of executing the *getLine* command—when doing this in your program, you will have to type `<-`. 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 $\mathit{readFile} :: \mathit{FilePath} \rightarrow \mathit{IO}\ \mathit{String}$, and then print the result to standard output:

$\mathbf{do}\ s \leftarrow \mathit{readFile}$ `"testFile.txt"`
$\quad \mathit{putStr}\ s$

> **Details:** Any type that is an instance of the *Monad* type class can be used with the **do** syntax to sequence actions. The *Monad* class is discussed in detail in Chapter 16. It suffices to say for now that the *IO* type is an instance of the *Monad* class.

15.3 Actions are Just Values

There are many other commands available for file, system, and user IO, some in the Standard Prelude, and some in various libraries (such as *IO*, *Directory*, *System*, and *Time*). We will not discuss many of these here, other than the MIDI IO commands described in Section 15.4.

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 do not *do* anything until they are sequenced appropriately using a **do** expression, and then returned as the value of the overall program (either as the variable *main* in the module *Main*, or typed at the GHCi prompt). Still, it is often convenient to place actions into a list as above, and the Haskell provides some useful functions for turning them into single commands. In particular, the function *sequence_* in the Standard Prelude, when used with IO, has type:

$$sequence\_ :: [IO\ a] \to IO\ ()$$

and can thus be applied to the *actionList* above to yield the single command:

$$main :: IO\ ()$$
$$main = sequence\_\ actionList$$

For a more interesting example of this idea, we first note that Haskell's strings are really just *lists of characters*. Indeed, *String* is a type synonym for a list of characters:

$$\textbf{type}\ String = [Char]$$

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 created using other type synonyms.)

Now back to the example. From the function $putChar :: Char \to IO\ ()$, 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:

$$putCharList :: String \to [IO\ ()]$$
$$putCharList = map\ putChar$$

With this, *putStr* is easily defined:

$$putStr :: String \to IO\ ()$$
$$putStr = sequence\_ \circ putCharList$$

Or, more succinctly:

$$putStr :: String \to IO\ ()$$
$$putStr = sequence\_ \circ map\ putStr$$

Of course, *putStr* can also be defined directly as a recursive function, which we do here just to emphasize that actions are just values, so we can use all of the functional programming skills that we normally use:

$$
\begin{array}{ll}
\mathit{putStr} & :: \mathit{String} \rightarrow \mathit{IO}\ () \\
\mathit{putStr}\ [\,] & = \mathit{return}\ () \\
\mathit{putStr}\ (c : cs) & = \textbf{do}\ \mathit{putChar}\ c \\
 & \phantom{= \textbf{do}}\ \mathit{putStr}\ cs
\end{array}
$$

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*, to be revealed in full in Chapter 16.

15.4 Reading and Writing MIDI Files

[TODO: Explain MIDI-file IO functions defined in *Codec.Midi*, as well as the Euterpea functions for writing MIDI files.]

Chapter 16

Higher-Order Types and Monads

All of the types that we have considered thus far in this text have been *first order*. For example, the type constructor *Music* has so far always been paired with an argument, as in *Music Pitch*. This is because *Music* by itself is a *type constructor*: something that takes a type as an argument and returns a type as a result. There are no *values* in Haskell that have this type, but such "higher-order types" can be used in type class declarations in useful ways, as we shall see in this chapter.

16.1 The Functor Class

To begin, consider the *Functor* class described previously in Section 7.4.3, and defined in the Standard Prelude:[1]

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

[1] The term *functor* (as well as the term *monad* to be introduced shortly) comes from a branch of abstract mathematics known as *category theory* [Pie91]. This reflects the strong mathematical principles that underly Haskell, but otherwise does not concern us here; i.e., you do not need to know anything about category theory to understand Haskell's functors and monads.

> **Details:** Type applications are written in the same manner as function applications, and are also left associative: the type $T\ a\ b$ is equivalent to $((T\ a)\ b)$.

There is something new here: the type variable f is applied to other type variables, as in $f\ a$ and $f\ b$. Thus we would expect f to be a *type constructor* such as *Music* that can be applied to an argument. Indeed, a suitable instance of *Functor* for *Music* is:

```
instance Functor Music where
  fmap f m = mMap f m
```

Similarly for *Primitive*:

```
instance Functor Primitive where
  fmap f p = pMap f p
```

Indeed, in retrospect, back in Chapter 6 where we defined *mMap* and *pMap*, we could have declared *Music* and *Primitive* as instances of *Monad* directly, and avoided defining the names *mMap* and *pMap* altogether:

```
instance Functor Music where
  fmap f (Prim p)       = Prim (fmap f p)
  fmap f (m1 :+: m2)    = fmap f m1 :+: fmap f m2
  fmap f (m1 :=: m2)    = fmap f m1 :=: fmap f m2
  fmap f (Modify c m)   = Modify c (fmap f m)
instance Functor Primitive where
  pMap              :: (a -> b) -> Primitive a -> Primitive b
  pMap f (Note d x) = Note d (f x)
  pMap f (Rest d)   = Rest d
```

In Haskell we write *Music Pitch* for a *Music* value instantiated on *Pitch* values; *Music* is the type constructor. Similarly, we write $[Int]$ for lists instantiated on integers; but what is the type constructor for lists? Because of Haskell's special syntax for the list data type, there is also a special syntax for its type constructor, namely $[\,]$.

> **Details:** Similarly, for tuples the type constructors are $(,)$, $(,,)$, $(,,,)$, and so on, and the type constructor for the function type is $(\rightarrow)$. This means that the following pairs of types are equivalent: $[a]$ and $[\,]\ a$, $f \rightarrow g$ and $(\rightarrow)\ f\ g$, (a, b) and $(,)\ a\ b$, and so on.

This allows us to create an instance of *Functor* for lists, as follows:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where} \\
\quad \mathit{fmap}\ f\ [\,] \qquad\;\; = [\,] \\
\quad \mathit{fmap}\ f\ (x : \mathit{xs}) = f\ x : \mathit{fmap}\ f\ \mathit{xs}
\end{array}$$

Note the use of $[\,]$ here in two ways: as a value in the list data type, and as a type constructor as described above.

Of course, the above declaration is equivalent to:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where} \\
\quad \mathit{fmap} = \mathit{map}
\end{array}$$

where *map* is the familiar function that we have been using since Chapter 3. This instance is in fact predefined in the Standard Prelude.

One of the nice things about the *Functor* class, of course, is that we can now use the same name, *fmap*, for lists, *Music*, and *Primitive* values (and any other data type for which an instance of *Functor* is declared). This could not have been done without higher-order type constructors, and here demonstrates the ability to handle generic "container" types, allowing functions such as *fmap* to work uniformly over them.

As mentioned in Section 7.7, type classes often imply a set of *laws* which govern the use of the operators in the class. In the case of the *Functor* class, the following laws are expected to hold:

$$\begin{array}{l}
\mathit{fmap}\ \mathit{id} \qquad\;\; = \mathit{id} \\
\mathit{fmap}\ (f \circ g) = \mathit{fmap}\ f \circ \mathit{fmap}\ g
\end{array}$$

Details: id is the *identity function*, $\lambda x \rightarrow x$. Although id is polymorphic, note that if its type on the left-hand side of the equation above is $a \rightarrow a$, then its type on the right must be $t\ a \rightarrow t\ a$, for some type constructor t that is an instance of $\mathit{Functor}$.

These laws ensure that the shape of the "container type" is unchanged by *fmap*, and that the contents of the container are not re-arranged by the mapping function.

Exercise 16.1 Verify that the instances of *Functor* for lists, *Primitive*, and *Music* are law-abiding.

16.2 The *Monad* Class

There are several classes in Haskell that are related to the notion of a monad, which can be viewed as a generalization of the principles that underly IO. Because of this, although the names of the classes and methods may seem unusual, these "monadic" operations are rather intuitive and useful for general programming.[2]

There are three classes associated with monads: *Functor* (which we have discussed already), *Monad* (also defined in the Standard Prelude), and *MonadPlus* (defined in *Control.Monad*).

The *Monad* class defines four basic operators: $(\gg\!=)$ (often pronounced "bind"), $(\gg)$ (often pronounced "sequence"), *return*, and *fail*:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad (\gg\!=) \ \ :: m\ a \to (a \to m\ b) \to m\ b \\
\quad (\gg) \ \ \ \ :: m\ a \to m\ b \to m\ b \\
\quad \mathit{return} :: a \to m\ a \\
\quad \mathit{fail} \ \ \ \ \ :: \mathit{String} \to m\ a \\
\quad m \gg k = m \gg\!= \backslash\_ \to k \\
\quad \mathit{fail}\ s \ \ \ = \mathit{error}\ s
\end{array}
$$

Details: The two infix operators above are typeset nicely here; using a text editor, you will have to type >>= and >> instead.

The default methods for $(\gg)$ and *fail* define behaviors that are almost always just what is needed. Therefore most instances of *Monad* need only define $(\gg\!=)$ and *return*.

Before studying examples of particular instances of *Monad*, we will first reveal another secret in Haskell, namely that the **do** syntax is actually shorthand for use of the monadic operators! The rules for this are a bit more involved than those for other syntax we have seen, but are still straightforward. The first rule is this:

$$\textbf{do}\ e \Rightarrow e$$

[2]Moggi [Mog89] was one of the first to point out the value of monads in describing the semantics of programming languages, and Wadler first popularized their use in functional programming [Wad92, PJW93].

So an expression such as **do** *putStr* `"Hello World"` is equivalent to just *putStr* `"Hello World"`.

The next rule is:

$$\begin{array}{l}\textbf{do } e_1; e_2; ...; en \\ \Rightarrow e_1 \gg \textbf{do } e_2; ...; en\end{array}$$

For example, combining this rule with the previous one means that:

$$\begin{array}{l}\textbf{do } \mathit{writeFile} \texttt{ "testFile.txt" "Hello File System"} \\ \quad \mathit{putStr} \texttt{ "Hello World"}\end{array}$$

is equivalent to:

$$\begin{array}{l}\mathit{writeFile} \texttt{ "testFile.txt" "Hello File System"} \gg \\ \mathit{putStr} \texttt{ "Hello World"}\end{array}$$

Note now that the sequencing of two commands is just the application of the function $(\gg)$ to two values of type $IO\ ()$. There is no magic here—it is all just functional programming!

> **Details:** What is the type of $(\gg)$ above? From the type class declaration we know that its most general type is:
>
> $$(\gg) :: Monad\ m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b$$
>
> However, in the case above, its two arguments both have type $IO\ ()$, so the type of $(\gg)$ must be:
>
> $$(\gg) :: IO\ () \rightarrow IO\ () \rightarrow IO\ ()$$
>
> That is, $m = IO$, $a = ()$, and $b = ()$. Thus the type of the result is $IO\ ()$, as expected.

The rule for pattern matching is the most complex, because we must deal with the situation where the pattern match fails:

$$\begin{array}{l}\textbf{do } pat \leftarrow e_1; e_2; ...; en \\ \Rightarrow \textbf{let } ok\ pat = \textbf{do } e_2; ...; en \\ \qquad\quad ok\ \_ \quad\ = fail \texttt{ "..."} \\ \quad\ \textbf{in } e_1 \gg\!= ok\end{array}$$

The right way to think of $(\gg\!=)$ above is simply this: it "executes" e_1, and then applies *ok* to the result. What happens after that is defined by *ok*: if the match succeeds, the rest of the commands are executed, otherwise the operation *fail* in the monad class is called, which in most cases (because of the default method) results in an *error*.

> **Details:** The string argument to *error* is a compiler-generated error message, preferably giving some indication of the location of the pattern-match failure.

A special case of the above rule is the case where the pattern *pat* is just a name, in which case the match cannot fail, so the rule simplifies to:

$$\begin{array}{l} \mathbf{do}\ x \leftarrow e_1; e_2; ...; en \\ \ \Rightarrow e_1 \gg\!= \lambda x \rightarrow \mathbf{do}\ e_2; ...; en \end{array}$$

The final rule deals with the **let** notation within a **do** expression:

$$\begin{array}{l} \mathbf{do\ let}\ \mathit{decllist}; e_2; ...; en \\ \ \Rightarrow \mathbf{let}\ \mathit{decllist}\ \mathbf{in\ do}\ e_2; ...; en \end{array}$$

> **Details:** Although we have not used this feature, note that a **let** inside of a **do** can take multiple definitions, as implied by the name *decllist*.

As mentioned earlier, because you already understand Haskell IO, you should have a fair amount of intuition about what the monadic operators do. Unfortuantely, we cannot look very closely at the instance of *Monad* for the type *IO*, since it ultimately relies on the state of the underlying operating system, which we do not have direct access to other than through primitive operations that communicate with it. Even then, these operations vary from system to system.

Nevertheless, a proper implementation of IO in Haskell is obliged to obey the following *monad laws*:

$$\begin{array}{ll} \mathit{return}\ a \gg\!= k & = k\ a \\ m \gg\!= \mathit{return} & = m \\ m \gg\!= (\lambda x \rightarrow k\ x \gg\!= h) & = (m \gg\!= k) \gg\!= h \end{array}$$

The first of these laws expresses the fact that *return* simply "sends" its value to the next action. Likewise, the second law says that if we immediately return the result of an action, we might as well just let the action return the value itself. The third law is the most complex, and essentially expresses an *associativity* property for the bind operator ($\gg\!=$). A special case of this law applies to the sequence operator ($\gg$):

$$m_1 \gg (m_2 \gg m_3) = (m_1 \gg m_2) \gg m_3$$

in which case the associativity is more obvious.

There is one other monad law, whose purpose is to connect the *Monad* class to the *Functor* class, and therefore only applies to types that are instances of both:

$$\mathit{fmap}\ f\ \mathit{xs} = \mathit{xs} \gg\!\!= \mathit{return} \circ f$$

We will see an example of this shortly.

Of course, this law can also be expressed in **do** notation:

$$\mathit{fmap}\ f\ \mathit{xs} = \mathbf{do}\ x \leftarrow \mathit{xs}; \mathit{return}\ (f\ x)$$

as can the previous ones for **do**:

$$\begin{aligned}
&\mathbf{do}\ x \leftarrow \mathit{return}\ a; k\ x &&= k\ a \\
&\mathbf{do}\ x \leftarrow m; \mathit{return}\ x &&= m \\
&\mathbf{do}\ x \leftarrow m; y \leftarrow k\ x; h\ y &&= \mathbf{do}\ y \leftarrow (\mathbf{do}\ x \leftarrow m; k\ x); h\ y \\
&\mathbf{do}\ m_1; m_2; m_3 &&= \mathbf{do}\ (\mathbf{do}\ m_1; m_2); m_3
\end{aligned}$$

So something like this:

```
do k ← getKey w
   return k
```

is equivalent to just *getKey w*, according to the second law above. As a final example, the third law above allows us to transform this:

```
do k ← getKey w
   n ← changeKey k
   respond n
```

into this:

```
let keyStuff = do k ← getKey w
                  changeKey k
in do n ← keyStuff
      respond n
```

Exercise 16.2 Verify the associativity law for $(\gg)$, starting with the associativity law for $(\gg\!\!=)$.

16.2.1 Other Instances of Monad

Maybe In addition to *IO*, the Standard Prelude's *Maybe* data type is a predefined instance of *Monad*:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{Just}\ x \quad\ \gg\!= k = k\ x \\
\quad \mathit{Nothing} \gg\!= k = \mathit{Nothing} \\
\quad \mathit{return} \qquad\quad = \mathit{Just} \\
\quad \mathit{fail}\ s \qquad\quad\ = \mathit{Nothing}
\end{array}$$

Details: *Maybe* is also a predefined instance of *Functor*:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \textbf{where} \\
\quad \mathit{fmap}\ f\ \mathit{Nothing} = \mathit{Nothing} \\
\quad \mathit{fmap}\ f\ (\mathit{Just}\ x) = \mathit{Just}\ (f\ x)
\end{array}$$

When used with this instance, the types of the monad operators are:

$$\begin{array}{l}
(\gg\!=)\ \ :: \mathit{Maybe}\ a \to (a \to \mathit{Maybe}\ b) \to \mathit{Maybe}\ b \\
\mathit{return} :: a \to \mathit{Maybe}\ a
\end{array}$$

We leave as an exercise the task of proving that this instance is law-abiding.

To see how this might be used, consider a computation involving functions $f :: \mathit{Int} \to \mathit{Int}$, $g :: \mathit{Int} \to \mathit{Int}$, and $x :: \mathit{Int}$:

$$g\ (f\ x)$$

Now suppose that each of the calculations using f and g could in fact be erroneous, and thus the results are encoded using the *Maybe* data type. Unfortunately this can become rather tedious to program, since each result that might be an error must be checked manually, as in:

$$\begin{array}{l}
\textbf{case}\ (f\ x)\ \textbf{of} \\
\quad \mathit{Nothing} \to \mathit{Nothing} \\
\quad \mathit{Just}\ y \quad\ \to \textbf{case}\ (g\ y)\ \textbf{of} \\
\qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{Nothing} \\
\qquad\qquad\qquad\quad \mathit{Just}\ z \quad\ \to z
\end{array}$$

Alternatively, we could take advantage of *Maybe*'s membership in the *Monad* class, and convert this into monadic form:

$$\begin{array}{l}
f\ x \gg\!= \lambda y \to \\
g\ y \gg\!= \lambda z \to
\end{array}$$

$\quad return\ z$

Or, using the more familiar **do** notation:

$$
\begin{array}{l}
\mathbf{do}\ y \leftarrow f\ x \\
\quad\ \ z \leftarrow g\ y \\
\quad\ \ return\ z
\end{array}
$$

Thus the tedium of the error check is "hidden" within the monad. In this sense monads are a good example of the abstraction principle in action (pardon the pun)!

It is also worth noting the following simplification:

$$
\begin{array}{l}
f\ x \gg\!= \lambda y \rightarrow \\
g\ y \gg\!= \lambda z \rightarrow \\
return\ z \\
\ \Rightarrow \{\mathit{currying\ simplification}\} \\
f\ x \gg\!= \lambda y \rightarrow \\
g\ y \gg\!= return \\
\ \Rightarrow \{\mathit{monad\ law\ for\ return}\} \\
f\ x \gg\!= \lambda y \rightarrow \\
g\ y \\
\ \Rightarrow \{\mathit{currying\ simplification}\} \\
f\ x \gg\!= g
\end{array}
$$

So we started with $g\ (f\ x)$ and ended with $f\ x \gg\!= g$; this is not too bad considering the alternative that we started with!

For an even more pleasing result, we can define a monadic composition operator:

$$
\begin{array}{l}
composeM :: Monad\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c) \\
(g\ `composeM`\ f)\ x = f\ x \gg\!= g
\end{array}
$$

in which case we started with $(g \circ f)\ x$ and ended with $(g\ `composeM`\ f)\ x$.

> **Details:** Note the type of *composeM*. It demonstrates that higher-order type constructors are also useful in type signatures.

Lists The list data type in Haskell is also a predefined instance of class *Monad*:

$$
\begin{array}{l}
\mathbf{instance}\ Monad\ [\,]\ \mathbf{where} \\
\quad m \gg\!= k\ = concat\ (map\ k\ m)
\end{array}
$$

$$\begin{aligned}&\mathit{return}\ x = [x]\\&\mathit{fail}\ x \quad\ = [\,]\end{aligned}$$

Details: Recall that *concat* takes a list of lists and concatenates them all together. It is defined in the Standard Prelude as:

$$\begin{aligned}&\mathit{concat} :: [[a]] \to [a]\\&\mathit{concat}\ \mathit{xss} = \mathit{foldr}\ (+\!\!+)\ [\,]\ \mathit{xss}\end{aligned}$$

The types of the monadic operators in this case are:

$$\begin{aligned}&(\gg\!=) :: [a] \to (b \to [b]) \to [b]\\&\mathit{return} :: a \to [a]\end{aligned}$$

The monadic functions in this context can be thought of as dealing with "multiple values." Monadic binding takes a set (list) of values and applies a function to each of them, collecting all generated values together. The *return* function creates a singleton list, and *fail* an empty one. For example,

$$\begin{aligned}&\mathbf{do}\ x \leftarrow [1,2,3]\\&\quad\ \ y \leftarrow [4,5,6]\\&\quad\ \ \mathit{return}\ (x,y)\end{aligned}$$

returns the list:

$$[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]$$

which happens to be the same list generated by:

$$[(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5,6]]$$

So list comprehension syntax is in essence another kind of monad syntax; indeed, they are not very different! (However, list comprehensions can only be used with lists.)

Note that if:

$$\mathbf{do}\ x \leftarrow \mathit{xs}; \mathit{return}\ (f\ x)$$

is equivalent to:

$$[f\ x \mid x \leftarrow \mathit{xs}]$$

(which is clearly just *map f xs*), then at least for the instance of lists in *Monad*, the last monad law makes perfect sense:

$$\mathit{fmap}\ f\ \mathit{xs} = \mathbf{do}\ x \leftarrow \mathit{xs}; \mathit{return}\ (f\ x)$$

Also note that the *Maybe* data type in monadic form behaves as a sort of truncated list in monadic form: *Nothing* is the same as [] and *Just x* is the same as [x].)

Exercise 16.3 Verify that all of the instance declarations in this section are law-abiding.

Exercise 16.4 Consider the *identity data type* defined by:

data *Id a* = *Id a*

Create an instance of *Monad* for *Id*, and prove that it is law-abiding.

16.2.2 Other Monadic Operations

The Standard Prelude has several functions specifically designed for use with monads; they are shown in Figure 16.1. Indeed, one of these we have already used: *sequence_*. Any mystery about how it works should be gone now; it is a very simple fold of the sequencing operator ($\gg$), with *return* () at the end. Note also the definition of *sequence*, a generalization of *sequence_* that returns a list of values of the intermediate results.

Finally, recall from Section 15.3 that *putStr* can be defined as:

```
putStr   :: String → IO ()
putStr s = sequence_ (map putChar s)
```

Using *mapM_* from Figure 16.1, this can be rewritten as:

```
putStr   :: String → IO ()
putStr s = mapM_ putChar s
```

16.3 The MonadPlus Class

The class *MonadPlus*, defined in the Standard Library *Control.Monad*, is used for monads that have a *zero element* and a *plus operator*:

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

The zero element should obey the following laws:

$$
\begin{array}{lll}
sequence & :: & Monad\ m \Rightarrow [m\ a] \rightarrow m\ [a] \\
sequence & = & foldr\ mcons\ (return\ [\,]) \\
 & & \quad \textbf{where}\ mcons\ p\ q = \textbf{do}\ \begin{array}[t]{l} x\ \leftarrow p \\ xs \leftarrow q \\ return\ (x : xs) \end{array} \\
sequence\_ & :: & Monad\ m \Rightarrow [m\ a] \rightarrow m\ () \\
sequence\_ & = & foldr\ (\gg)\ (return\ ()) \\
mapM & :: & Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b] \\
mapM\ f\ as & = & sequence\ (map\ f\ as) \\
mapM\_ & :: & Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ () \\
mapM\_\ f\ as & = & sequence\_\ (map\ f\ as) \\
(=\!\ll) & :: & Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow m\ a \rightarrow m\ b \\
f =\!\ll x & = & x \gg\!= f
\end{array}
$$

Figure 16.1: Monadic Utility Functions

$$
\begin{array}{ll}
m \gg\!= (\lambda x \rightarrow mzero) & = mzero \\
mzero \gg\!= m & = mzero
\end{array}
$$

and the plus operator should obey these:

$$
\begin{array}{l}
m\ \text{`}mplus\text{`}\ mzero = m \\
mzero\ \text{`}mplus\text{`}\ m = m
\end{array}
$$

By analogy to arithmetic, think of *mzero* as 0, *mplus* as addition, and $(\gg\!=)$ as multiplication. The above laws should then make more sense.

For the *Maybe* data type, the zero and plus values are:

$$
\begin{array}{l}
\textbf{instance}\ MonadPlus\ Maybe\ \textbf{where} \\
\quad \begin{array}{ll}
mzero & = Nothing \\
Nothing\ \text{`}mplus\text{`}\ ys & = ys \\
xs\ \quad\quad \text{`}mplus\text{`}\ ys & = xs
\end{array}
\end{array}
$$

and for lists they are:

$$
\begin{array}{l}
\textbf{instance}\ MonadPlus\ [\,]\ \textbf{where} \\
\quad mzero = [\,] \\
\quad mplus = (+\!\!+)
\end{array}
$$

So you can see now that the familiar concatentation operation $(+\!\!+)$ that we have been using all along for lists is just a special case of the *mplus* operator.

It is worth pointing out that the IO monad is not an instance of the *MonadPlus* class, since it has no zero element. For if it did have a zero element, then the IO action *putStr* `"Hello"` $\gg$ *zero* should *not* print the string `"Hello"`, according to the first zero law above. But this is counter-intuitive, or at least is certainly not what the designers of Haskell had in mind for IO.

The *Monad* module in the Standard Library also includes several other useful functions defined in terms of the monadic primitives. You are encouraged to read these for possible use in your own programs.

Exercise 16.5 Verify that the instances of *MonadPlus* for the *Maybe* and list data types are law-abiding.

16.4 State Monads

Monads are commonly used to simulate stateful, or imperative, computations, in which the details of updating and passing around the state are hidden within the mechanics of the monad. Generally speaking, a *state monad* has a type of the form:

$$\textbf{data}\ SM\ s\ a = SM\ (s \to (s, a))$$

where s is the state type, and a is the value type. The instance of this type in *Monad* is given by:

$$
\begin{array}{l}
\textbf{instance}\ Monad\ (SM\ s)\ \textbf{where} \\
\quad return\ a \\
\quad\quad = SM\ \$\ \lambda s_0 \to (s_0, a) \\
\quad SM\ sm_0 \ggg\!= fsm_1 \\
\quad\quad = SM\ \$\ \lambda s_0 \to \\
\quad\quad\quad \textbf{let}\ (s_1, a_1) \quad = sm_0\ s_0 \\
\quad\quad\quad\quad\quad SM\ sm_1 = fsm_1\ a_1 \\
\quad\quad\quad\quad\quad (s_2, a_2) \quad = sm_1\ s_1 \\
\quad\quad\quad \textbf{in}\ (s_2, a_2)
\end{array}
$$

The last equation in the **let** expression could obviously be eliminated, but it is written this way to stress the symmetry in the treatment of the two commands.

> **Details:** Note that SM is a type constructor that takes *two* type arguments. Applyiing it to one argument (as in $SM\ s$ above) is a kind of type-level currying, yielding a new type constructor that takes one argument, as required by the $Monad$ class.

A good example of a state monad, at least abstractly speaking, is Haskell's IO type, where the state s can be thought of as the "state of the world," such as the contents of the file system, the image on a display, and the output of a printer.

But what about creating our own state monad? As a simple example, consider this definition of a $Tree$ data type:

$$\begin{array}{l}\textbf{data}\ Tree\ a = Leaf\ a \mid Branch\ (Tree\ a)\ (Tree\ a)\\ \quad \textbf{deriving}\ Show\end{array}$$

Suppose now we wish to define a function $label :: Tree\ a \to Tree\ Int$ such that, for example, the value $test$:

$$\begin{array}{l}test = \textbf{let}\ t = Branch\ (Leaf\ \texttt{'a'})\ (Leaf\ \texttt{'b'})\\ \quad \textbf{in}\ label\ (Branch\ t\ t)\end{array}$$

evaluates to:

$$\begin{array}{l}Branch\ (Branch\ (Leaf\ 0)\ (Leaf\ 1))\\ \qquad\qquad (Branch\ (Leaf\ 2)\ (Leaf\ 3))\end{array}$$

Without knowing anything about monads, this job is relatively easy:

$$\begin{array}{l}label\ \ :: Tree\ a \to Tree\ Int\\ label\ t = snd\ (lab\ t\ 0)\\[4pt] lab :: Tree\ a \to Int \to (Int, Tree\ Int)\\ lab\ (Leaf\ a)\ n\\ \quad = (n+1, Leaf\ n)\\ lab\ (Branch\ t_1\ t_2)\ n\\ \quad = \textbf{let}\ (n_1, t'_1) = lab\ t_1\ n\\ \qquad\qquad\ (n_2, t'_2) = lab\ t_2\ n_1\\ \qquad\ \textbf{in}\ \ (n_2, Branch\ t'_1\ t'_2)\end{array}$$

Although simple, there is an undeniable tedium in "threading" the value of n from one call to lab to the next. To solve this problem, note that $lab\ t$ has type $Int \to (Int, Tree\ Int)$, which is in the right form for a state monad. Of course, we need a true data type, and so we write:

$$\textbf{newtype}\ Label\ a = Label\ (Int \to (Int, a))$$

Details: A **newtype** declaration behaves just like a **data** declaration, except that only one constructor is allowed on the right-hand side. This allows the compiler to implement the datatype more efficiently, since it "knows" that only one possibility exists. It is also more type-safe than a type synonym, since, like **data**, it generates a new type, rather than being a synonym for an existing type.

The *Monad* instance for *Label* is just like that for *SM* above:

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad\ Label} \textbf{ where} \\
\quad \mathit{return}\ a \\
\quad\quad = \mathit{Label}\ \$\ \lambda s \rightarrow (s, a) \\
\quad \mathit{Label}\ \mathit{lt}_0 \gg\!= \mathit{flt}_1 \\
\quad\quad = \mathit{Label}\ \$\ \lambda s_0 \rightarrow \\
\quad\quad\quad \textbf{let } (s_1, a_1) \quad = \mathit{lt}_0\ s_0 \\
\quad\quad\quad\quad\quad \mathit{Label}\ \mathit{lt}_1 = \mathit{flt}_1\ a_1 \\
\quad\quad\quad \textbf{in } \mathit{lt}_1\ s_1
\end{array}
$$

Whereas the monad handles the threading of the state, we also need a way to extract information from the state, as needed in a particular application. In the case of labeling trees, we need to know what the current value of the state (an *Int*) is, at each point that we encounter a leaf. So we define:

$$
\begin{array}{l}
\mathit{getLabel} :: \mathit{Label\ Int} \\
\mathit{getLabel} = \mathit{Label}\ \$\ \lambda n \rightarrow (n + 1, n)
\end{array}
$$

Now we can write the following monadic version of the labeling function:

$$
\begin{array}{l}
\mathit{mlabel} :: \mathit{Tree}\ a \rightarrow \mathit{Tree\ Int} \\
\mathit{mlabel}\ t = \textbf{let } \mathit{Label}\ \mathit{lt} = \mathit{mlab}\ t \\
\quad\quad\quad\quad\quad \textbf{in } \mathit{snd}\ (\mathit{lt}\ 0)
\end{array}
$$

$$
\begin{array}{l}
\mathit{mlab} :: \mathit{Tree}\ a \rightarrow \mathit{Label}\ (\mathit{Tree\ Int}) \\
\mathit{mlab}\ (\mathit{Leaf}\ a) \\
\quad = \textbf{do } n \leftarrow \mathit{getLabel} \\
\quad\quad\quad\quad \mathit{return}\ (\mathit{Leaf}\ n) \\
\mathit{mlab}\ (\mathit{Branch}\ t_1\ t_2) \\
\quad = \textbf{do } t'_1 \leftarrow \mathit{mlab}\ t_1 \\
\quad\quad\quad\quad t'_2 \leftarrow \mathit{mlab}\ t_2 \\
\quad\quad\quad\quad \mathit{return}\ (\mathit{Branch}\ t'_1\ t'_2)
\end{array}
$$

Note that the threading of the state has been completely eliminated from *mlab*, as has the incrementing of the state, which has been isolated in the function *getLabel*.

As an example, this test case:

$$\begin{aligned} mtest = \mathbf{let}\ & t = Branch\ (Leaf\ \texttt{'a'})\ (Leaf\ \texttt{'b'}) \\ \mathbf{in}\ & mlabel\ (Branch\ t\ t) \end{aligned}$$

generates the same result as the non-monadic version above.

For this simple example you may decide that eliminating the threading of state is not worth it. Indeed, in reality it has just been moved from the definition of *lab* to the method declaration for $(\gg=)$, and the new version of the program is certainly longer than the old! But the capture of repetitious code into one function is the whole point of the abstraction principle, and hopefully you can imagine a context where threading of state happens often, perhaps hundreds of times, in which case the abstraction will surely pay off. IO is one example of this (imagine threading the state of the world on every IO command).

Exercise 16.6 Recall the definition of *replFun* in Chapter **??**, Section 12.2. Note how it threads the random number source through the program. Rewrite this function using a state monad so that this threading is eliminated.

16.5 Type Class Type Errors

As you know, Haskell's type system detects ill-typed expressions. But what about errors due to malformed types? The value $(+)\ 1\ 2\ 3$ results in a type error since $(+)$ takes only two arguments. Similarly, the type *Tree Int Int* should result in some sort of an error since the *Tree* type constructor takes only a single argument. So, how does Haskell detect malformed types? The answer is a second type system which ensures the correctness of types! That is, each type is assigned its own type—which is called its *kind*—and these kinds are used to ensure that the type is used correctly.

There are only two kinds that we need to consider:

- The symbol $*$ represents the kind of type associated with concrete data objects. That is, if the value v has type t, then the kind of t must be $*$.
- If κ_1 and κ_2 are kinds, then $\kappa_1 \rightarrow \kappa_2$ is the kind of types that take a type of kind κ_1 and return a type of kind κ_2.

The details of how kinds are used to detect malformed types are beyond the scope of this text, but it is helpful to walk through a familiar example:

Int has kind $*$, as does the type *Tree Int*. The type constructor *Tree*, however, has kind $* \to *$. Instances of the *Functor* class must all have the kind $* \to *$. Thus a kind error would result from a declaration such as:

> **instance** *Functor Int* **where** ...

or

> **instance** *Functor* (*Tree Int*) **where** ...

Kinds do not appear directly in Haskell programs; the Haskell system infers them without any need for "kind declarations." Kinds stay in the background of a Haskell program except when a kind error occurs, in which case an error message may refer to the kind conflict. Fortunately, kinds are simple enough that your Haskell system should be able to provide descriptive error messages in most cases.

Chapter 17

Musical User Interface

Daniel Winograd-Cort

{-# LANGUAGE Arrows #-}
module *Euterpea.Examples.MUI* **where**
import *Euterpea*

This module is not part of the standard Euterpea module hierarchy (i.e. those modules that get imported by the header command "**import** *Euterpea*"), but it can be found in the *Examples* folder in the Euterpea distribution, and can be imported into another module by the header command:

import *Euterpea.Examples.MUI*

> **Details:** To use the *arrow syntax* described in this chapter, it is necessary to use the following compiler pragma in GHC:
>
> {-# LANGUAGE Arrows #-}

17.1 Introduction

Many music software packages have a graphical user interface (aka "GUI") that provides varying degrees of functionality to the user. In Euterpea a basic set of widgets is provided that are collectively referred to as the *mu-*

sical user interface, or MUI. This interface is quite different from the GUI interfaces found in most conventional languages, and is built around the concepts of *signal functions* and *arrows* [HCNP03, Hug00].[1] Signal functions are an abstraction of the time-varying values inherent in an interactive system such as a GUI or Euterpea's MUI. Signal functions are provided for creating graphical sliders, pushbuttons, and so on for input; textual displays, graphs, and graphic images for output; and textboxes, virtual keyboards, and more for combinations of input and output. In addition to these graphical widgets, the MUI also provides an interface to standard MIDI input and output devices.

17.2 Basic Concepts

A *signal* is a time-varying quantity. Conceptually, at least, most things in our world, and many things that we program with, are time-varying. The position of a mouse is time-varying. So is the voltage used to control a motor in a robot arm. Even an animation can be thought of as a time-varying image.

A *signal function* is an abstract function that converts one signal into another. Using the examples above, a signal function may add an offset to a time-varying mouse position, filter out noise from the time-varying voltage for a robot motor, or speed up or slow down an animation.

Perhaps the simplest way to understand Euterpea's approach to programming with signals is to think of it as a language for expressing *signal processing diagrams* (or equivalently, electrical circuits). We can think of the lines in a typical signal processing diagram as signals, and the boxes that convert one signal into another as signal functions. For example, this very simple diagram has two signals, x and y, and one signal function, *sigfun*:

Image not in repository!

Using Haskell's *arrow syntax* [Hug00, Pat01], this diagram can be expressed as a code fragment in Euterpea simply as:

$$y \leftarrow \mathit{sigfun} \prec x$$

[1] The Euterpea MUI is built using the arrow-based GUI library *UISF*, which is its own standalone package. UISF, in turn, borrows concepts from *Fruit* [CE01, Cou04].

Details: The syntax $\leftarrow$ and $\prec$ is typeset here in an attractive way, but the user will have to type `<-` and `-<`, respectively, in her source file.

In summary, the arrow syntax provides a convenient way to compose signal functions together—i.e. to wire together the boxes that make up a signal processing diagram.

17.2.1 The Type of a Signal Function

Polymorphically speaking, a signal function has type $SF\ a\ b$, which should be read, "the type of signal functions that convert signals of type a into signals of type b."

For example, suppose the signal function *sigfun* used earlier has type $SF\ T_1\ T_2$, for some types T_1 and T_2. In that case, and using the example above, x will have type T_1, and y will have type T_2. Although signal functions act on signals, the arrow notation allows us to manipulate the instantaneous values of the signals, such as x and y above, directly.

A signal function whose type is of the form $SF\ ()\ b$ essentially takes no input, but produces some output of type b. Because of this we often refer to such a signal function as a *signal source*. Similarly, a signal function of type $SF\ a\ ()$ is called a *signal sink*—it takes input, but produces no output. Signal sinks are essentially a form of output to the real world.

We can also create and use signal functions that operate on signals of tuples. For example, a signal function $exp :: SF\ (Double, Double)\ Double$ that raises the first argument in a tuple to the power of its second, at every point in time, could be used as follows:

$$z \leftarrow exp \prec (x, y)$$

As mentioned earlier, a signal function is "abstract," in the sense that it cannot be applied like an ordinary function. Indeed, *SF* is an instance of the *Arrow* type class in Haskell, which only provides operations to *compose* one signal function with another in several ways. The *Arrow* class and how all this works for signal functions will be described in Chapter **??**. For now, it suffices to say that programming in this style can be awkward and that Haskell provides the arrow syntax described above to make the programming easier and more natural.

A Euterpea MUI program expresses the composition of a possibly large number of signal functions into a composite signal function that is then "run" at the top level by a suitable interpreter. A good analogy for this idea is a state or IO monad, where the state is hidden, and a program consists of a linear sequencing of actions that are eventually run by an interpreter or the operating system. But in fact arrows are more general than monads, and in particular the composition of signal functions does not have to be completely linear, as will be illustrated shortly.

17.2.2 proc Declarations

Arrows and arrow syntax will be described in more detail in Chapter **??**. For now, keep in mind that $\leftarrow$ and $\prec$ are part of the *syntax*, and are not simply binary operators. Indeed, we cannot just write the earlier code fragments anywhere. They have to be within an enclosing **proc** construct whose result type is that of a signal function. The **proc** construct begins with the keyword **proc** along with a formal parameter, analogous to an anonymous function. For example, a signal function that takes a signal of type *Double* and adds 1 to it at every point in time, and then applies *sigfun* to the resulting signal, can be written:

$$
\begin{array}{l}
\mathbf{proc}\ y \rightarrow \mathbf{do} \\
\quad x \leftarrow \mathit{sigfun} \prec y + 1 \\
\quad \mathit{outA} \prec x
\end{array}
$$

outA is a special signal function that specifies the output of the signal function being defined.

> **Details:** The **do** keyword in arrow syntax introduces layout, just as it does in monad syntax.

Note the analogy of this code to the following snippet involving an ordinary anonymous function:

$$
\begin{array}{l}
\lambda y \rightarrow \\
\quad \mathbf{let}\ x = \mathit{sigfun}'\ (y + 1) \\
\quad \mathbf{in}\ x
\end{array}
$$

The important difference, however, is that *sigfun* works on a signal, i.e. a time-varying quantity. To make the analogy a little stronger, we could imagine a signal being implemented as a stream of discrete values. In which case, to achieve the effect of the arrow code given d, we would have to write something like this:

$$\begin{array}{l} \lambda ys \rightarrow \\ \quad \textbf{let } xs = sigfun'' \; (map \; (+1) \; ys) \\ \quad \textbf{in } xs \end{array}$$

The arrow syntax allows us to avoid worrying about the streams themselves.

17.2.3 Four Useful Functions

There are four useful auxiliary functions that will make writing signal functions a bit easier. The first two essentially "lift" constants and functions from the Haskell level to the arrow (signal function) level:

$$\begin{array}{ll} arr & :: (a \rightarrow b) \rightarrow SF \; a \; b \\ constA & :: \qquad\; b \;\; \rightarrow SF \; () \; b \end{array}$$

For example, a signal function that adds one to every sample of its input can be written simply as *arr* (+1), and a signal function that returns the constant 440 as its result can be written *constA* 440 (and is a signal source, as defined earlier).

The other two functions allow us to *compose* signal functions:

$$\begin{array}{l} (\ggg) :: SF \; a \; b \rightarrow SF \; b \; c \rightarrow SF \; a \; c \\ (\lll) :: SF \; b \; c \rightarrow SF \; a \; b \rightarrow SF \; a \; c \end{array}$$

$(\lll)$ is analogous to Haskell's standard composition operator $(\circ)$, whereas $(\ggg)$ is like "reverse composition."

As an example that combines both of the ideas above, recall the very first example given in this chapter:

$$\begin{array}{l} \textbf{proc } y \rightarrow \textbf{do} \\ \quad x \leftarrow sigfun \prec y + 1 \\ \quad outA \prec x \end{array}$$

which essentially applies *sigfun* to one plus the input. This signal function can be written more succinctly as either *arr* (+1) $\ggg$ *sigfun* or *sigfun* $\lll$ *arr* (+1).

The functions $(\ggg)$, $(\lll)$, and *arr* are actually generic operators on arrows, and thus to use them one may import them from the *Arrow* library. However, Euterpea reexports them automatically so we need not do this.

17.2.4 Events

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:

$$\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

Conceptually, we define an event simply as a value of type *Maybe a*, for some type a. We say that the value associated with an event is “attached to” or “carried by” that event.

However, to fit this into the signal function paradigm, we imagine *signals of events*—in other words, *event streams*. So a signal function that takes events of type $\mathit{Maybe}\ T_1$ as input, and emits events of type $\mathit{Maybe}\ T_2$, would have type $\mathit{SF}\ (\mathit{Maybe}\ T_1)\ (\mathit{Maybe}\ T_2)$. When there is no event, an event stream will have the instantaneous value *Nothing*, and when an event occurs, it will have the value $\mathit{Just}\ x$ for some value x.

For convenience Euterpea defines a type synonym for events:

$$\mathbf{type}\ \mathit{SEvent}\ a = \mathit{Maybe}\ a$$

The name *SEvent* is used to distinguish it from performance *Event* as defined in Chapter 8. “*SEvent*” can be read as “signal event.”

17.2.5 Feedback

If we think about signal functions and arrows as signal processing diagrams, then so far, we have only considered how to connect them so that the streams all flow in the same direction. However, there may be times that we want to feed an output of one signal function back in as one of its inputs, thus creating a loop.

How can a signal function depend on its own output? At some point in the loop, we need to introduce a *delay* function. Euterpea has a few different delay functions that we will decribe in more detail later in this chapter (Section 17.4.4), but for now, we will casually introduce the simplest of these: *fcdelay*.

$$\mathit{fcdelay} :: b \to \mathit{DeltaT} \to \mathit{SF}\ b\ b$$

The name *fcdelay* stands for “fixed continuous delay”, and it delays a continuous signal for a fixed amount of time. (Note that *DeltaT* is a type synonym for *Double* and represents a change in time, or δt.) Thus, the signal function *fcdelay* b t will delay its input signal for t seconds, emitting the constant signal b for the first t seconds.

With a delay at the ready, we can create a loop in a signal function by using the **rec** keyword in the arrow syntax. This keyword behaves much like it does in monadic **do** syntax and allows us to use a signal before we have defined it.

For instance, we can create a signal function that will count how many seconds have gone by since it started running:

$$
\begin{aligned}
&\mathit{secondCounter} :: \mathit{SF}\ ()\ \mathit{Integer} \\
&\mathit{secondCounter} = \mathbf{proc}\ () \rightarrow \mathbf{do} \\
&\quad \mathbf{rec}\ \mathit{count} \leftarrow \mathit{fcdelay}\ 0\ 1 \prec \mathit{count} + 1 \\
&\quad \mathit{outA} \prec \mathit{count}
\end{aligned}
$$

> **Details:** The **rec** keyword comes from an extension to arrows called *arrow loop*. To use the same ability outside of the arrow syntax requires the *loop* operator:
>
> $$\mathit{loop} :: \mathit{SF}\ (b, d)\ (c, d) \rightarrow \mathit{SF}\ b\ c$$

17.2.6 [Advanced] Why Arrows?

It is possible, and fairly natural, to define signal functions directly, say as an abstract type *Signal* T, and then define functions to add, multiply, take the sine of, and so on, signals represented in this way. For example, *Signal Float* would be the type of a time-varying floating-point number, *Signal AbsPitch* would be the type of a time-varying absolute pitch, and so on. Then given $s_1, s_2 :: \mathit{Signal\ Float}$ we might simply write $s_1 + s_2$, $s_1 * s_2$, and $\mathit{sin}\ s_1$ as examples of applying the above operations. Haskell's numeric type class hierarchy makes this particularly easy to do. Indeed, several domain-specific languages based on this approach have been designed, beginning with the language *Fran* [EH97] that was designed for writing computer animation programs.

But years of experience and theoretical study have revealed that such an approach leads to a language with subtle time- and space-leaks,[2] for reasons that are beyond the scope of this textbook [LH07].

[2] A time-leak in a real-time system occurs whenever a time-dependent computation falls behind the current time because its value or effect is not needed yet, but then requires

Perhaps surprisingly, these problems can be avoided by using arrows. Programming in this style gives the user access to signal functions, and the individual values that comprise a signal, but not to the actual signal itself. By not giving the user direct access to signals, and providing a disciplined way to compose signal functions (namely arrow syntax), time- and space-leaks are avoided. In fact, the resulting framework is highly amenable to optimization, although this requires using special features in Haskell, as described in Chapter **??**.

17.3 The UISF Arrow

SF as used in this chapter so far is an instance of the *Arrow* class, but is not the actual type used for constructing MUIs. The core component of Euterpea's MUI is the *user interface signal function*, captured by the type *UISF*, which is also an instance of the *Arrow* class. So instead of *SF*, in the remainder of this chapter we will use *UISF*, but all of the previous discussion about signal functions and arrows still applies.

Using *UISF*, we can create "graphical widgets" using a style very similar to the way we wired signal functions earlier. However, instead of having values of type $SF\ a\ b$, we will use values of type $UISF\ a\ b$. Just like *SF*, the *UISF* type is fully abstract (meaning its implementation is hidden) and, being an instance of the *Arrow* class, can be used with arrow syntax.

17.3.1 Graphical Input and Output Widgets

Euterpea's basic widgets are shown in Figure 17.1. Note that each of them is, ultimately, a value of type $UISF\ a\ b$, for some input type a and output type b, and therefore may be used with the arrow syntax to help coordinate their functionality. The names and type signatures of these functions suggest their functionality, which we elaborate in more detail below:

- A simple (static) text string can be displayed using:

  $$label :: String \rightarrow UISF\ a\ a$$

- Alternatively, a time-varying string can be be displayed using:

"catching up" at a later point in time. This catching up process can take an arbitrarily long time, and may consume additional space as well. It can destroy any hope for real-time behavior if not managed properly.

$$\begin{array}{ll}
\mathit{label} & :: \mathit{String} \rightarrow \mathit{UISF}\ a\ a \\
\mathit{displayStr} & :: \mathit{UISF}\ \mathit{String}\ () \\
\mathit{display} & :: \mathit{Show}\ a \Rightarrow \mathit{UISF}\ a\ () \\
\mathit{withDisplay} & :: \mathit{Show}\ b \Rightarrow \mathit{UISF}\ a\ b \rightarrow \mathit{UISF}\ a\ b \\
\mathit{textbox} & :: \mathit{UISF}\ \mathit{String}\ \mathit{String} \\
\mathit{textboxE} & :: \mathit{String} \rightarrow \mathit{UISF}\ (\mathit{SEvent}\ \mathit{String})\ \mathit{String} \\
\mathit{radio} & :: [\mathit{String}] \rightarrow \mathit{Int} \rightarrow \mathit{UISF}\ ()\ \mathit{Int} \\
\mathit{button} & :: \mathit{String} \rightarrow \mathit{UISF}\ ()\ \mathit{Bool} \\
\mathit{checkbox} & :: \mathit{String} \rightarrow \mathit{Bool} \rightarrow \mathit{UISF}\ ()\ \mathit{Bool} \\
\mathit{checkGroup} & :: [(\mathit{String}, a)] \rightarrow \mathit{UISF}\ ()\ [a] \\
\mathit{listbox} & :: (\mathit{Eq}\ a, \mathit{Show}\ a) \Rightarrow \mathit{UISF}\ ([a], \mathit{Int})\ \mathit{Int} \\
\mathit{hSlider}, \mathit{vSlider} & :: \mathit{RealFrac}\ a \Rightarrow (a, a) \rightarrow a \rightarrow \mathit{UISF}\ ()\ a \\
\mathit{hiSlider}, \mathit{viSlider} & :: \mathit{Integral}\ a \Rightarrow a \rightarrow (a, a) \rightarrow a \rightarrow \mathit{UISF}\ ()\ a
\end{array}$$

Figure 17.1: Basic MUI Input/Output Widgets

$\mathit{displayStr} :: \mathit{UISF}\ \mathit{String}\ ()$

For convenience, Euterpea defines the following useful variations of *displayStr*:

$$\begin{array}{l}
\mathit{display} :: \mathit{Show}\ a \Rightarrow \mathit{UISF}\ a\ () \\
\mathit{display} = \mathit{arr}\ \mathit{show} \ggg \mathit{displayStr} \\
\mathit{withDisplay} \quad :: \mathit{Show}\ b \Rightarrow \mathit{UISF}\ a\ b \rightarrow \mathit{UISF}\ a\ b \\
\mathit{withDisplay}\ \mathit{sf} = \mathbf{proc}\ a \rightarrow \mathbf{do} \\
\quad b \leftarrow \mathit{sf} \prec a \\
\quad \mathit{display} \prec b \\
\quad \mathit{outA} \quad \prec b
\end{array}$$

display allows us to display anything that is "*Show*able." *withDisplay* is an example of a *signal function transformer*: it takes a signal function and attaches a display widget to it that displays the value of its time-varying output.

- A textbox that functions for both input and output can be created using:

  $\mathit{textbox} :: \mathit{UISF}\ \mathit{String}\ \mathit{String}$

A *textbox* in Euterpea is notable because it is "bidirectional." That is, the time-varying input is displayed, and the user can interact with it by typing or deleting, the result being the time-varying output. In practice, the textbox is used almost exclusively with the **rec** keyword and a *delay* operator. For example, a code snippet from a MUI that uses *textbox* may look like this:

$$\mathbf{rec}\ str \leftarrow textbox \lll delay\ \texttt{"Initial text"} \prec str$$

Because of this common usage, there is a variant of the textbox:

$$textboxE :: String \rightarrow UISF\ (SEvent\ String)\ String$$

A *textboxE* widget encapsulates the recursion and delay internally. Thus, its initial value is given by its static argument, and its input stream is an event stream that will update the displayed text when there is an event and leave it unchanged otherwise.

- *radio*, *button*, and *checkbox* are three kinds of "pushbuttons." A *button* (or *checkbox*) is pressed and unpressed (or checked and unchecked) independently of others. In contrast, a *radio* button is dependent upon other radio buttons—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.

  The *checkGroup* widget creates a group of *checkbox*es. As its static argument, it takes a list of pairs of strings and values. For each pair, one *checkbox* is created with the associated string as its label. Rather than simply returning *True* or *False* for each checked box, it returns a list of the values associated with each label as its output stream.

- The *listbox* widget creates a pane with selectable text entries. The input stream is the list of entries as well as which entry is currently selected, and the output stream is the index of the newly selected entry. In many ways, this widget functions much like the *radio* widget except that it is stylistically different, it is dynamic, and, like the *textbox* widget, it is bidirectional.

- *hSlider*, *vSlider*, *hiSlider* and *viSlider* are four kinds of "sliders"—a graphical widget that looks like a slider control as found on a hardware device. The first two yield floating-point numbers in a given range, and are 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 simple example, here is a MUI that has a single slider representing absolute pitch, and a display widget that displays the pitch corresponding to the current setting of the slider:

$$\begin{array}{l} ui_0 :: UISF\ ()\ () \\ ui_0 = \mathbf{proc}\ \_ \rightarrow \mathbf{do} \\ \quad ap \leftarrow hiSlider\ 1\ (0,100)\ 0 \prec ()\\ \quad display \prec pitch\ ap \end{array}$$

Note how the use of signal functions makes this dynamic MUI trivial to write. But using the functions defines in Section 19.1.2 it can be defined even more succinctly as:

$$ui_0 = hiSlider\ 1\ (0,100)\ 0 \ggg arr\ pitch \ggg display$$

We can execute this example using the function:

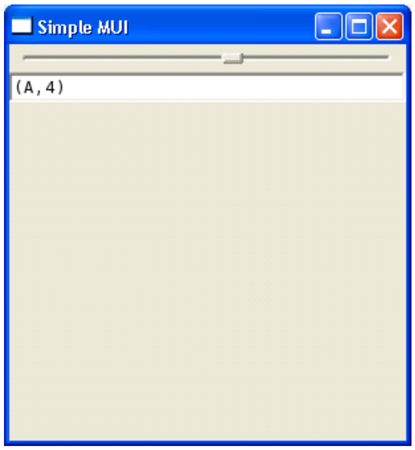
$$runMUI' :: UI\ ()\ () \rightarrow IO\ ()$$

So our first running example of a MUI is:

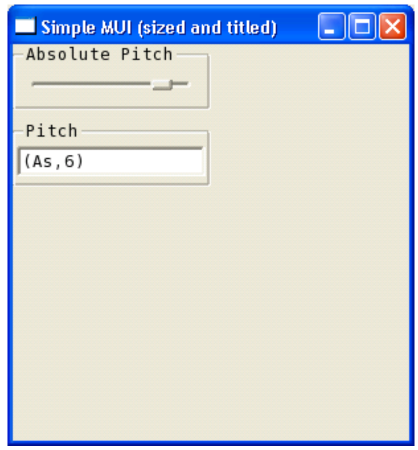
$$mui_0 = runMUI'\ ui_0$$

The resulting MUI, once the slider has been moved a bit, is shown in Figure 17.2(a).

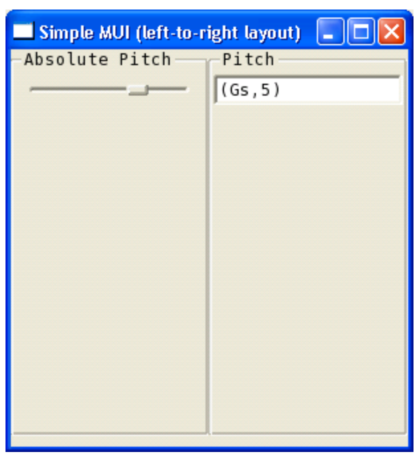
Details: Any MUI widgets have the capacity to be *focusable*, which is particularly relevant for graphical widgets. When a focusable widget is "in focus," not only can it update its appearance, but any key presses from the computer keyboard become visible to it as input events. This means that keyboard controls are possible with MUI widgets. Obviously, typing will affect the value of a textbox widget, but also, for instance, the arrow keys as well as the "Home" and "End" keys will affect the value of a slider widget. Focus can be shifted between widgets by clicking on them with the mouse as well as by using "Tab" and "Shift+Tab" to cycle focus through focusable widgets.



(a) Very Simple



(b) With Titles and Sizing



(c) With Alternate (left-to-right) Layout

Figure 17.2: Several Simple MUIs

$$
\begin{array}{lll}
\mathit{title} & :: \mathit{String} & \to \mathit{UISF}\ a\ b \to \mathit{UISF}\ a\ b \\
\mathit{setLayout} & :: \mathit{Layout} & \to \mathit{UISF}\ a\ b \to \mathit{UISF}\ a\ b \\
\mathit{setSize} & :: (\mathit{Int}, \mathit{Int}) & \to \mathit{UISF}\ a\ b \to \mathit{UISF}\ a\ b \\
\mathit{pad} & \multicolumn{2}{l}{:: (\mathit{Int}, \mathit{Int}, \mathit{Int}, \mathit{Int}) \to \mathit{UISF}\ a\ b \to \mathit{UISF}\ a\ b}
\end{array}
$$

$$\mathit{topDown}, \mathit{bottomUp}, \mathit{leftRight}, \mathit{rightLeft} :: \mathit{UISF}\ a\ b \to \mathit{UISF}\ a\ b$$

$$
\begin{array}{l}
\mathit{makeLayout} :: \mathit{LayoutType} \to \mathit{LayoutType} \to \mathit{Layout} \\
\textbf{data}\ \mathit{LayoutType} = \mathit{Stretchy}\ \{\mathit{minSize} :: \mathit{Int}\} \\
\qquad\qquad\qquad\quad \mid\ \mathit{Fixed}\quad \{\mathit{fixedSize} :: \mathit{Int}\}
\end{array}
$$

Figure 17.3: MUI Layout Widget Transformers

17.3.2 Widget Transformers

Figure 17.3 shows a set of "widget transformers"—functions that take UISF values as input, and return modified UISF values as output.

- *title* simply attaches a title (a string) to a UISF, and *setLayout* establishes a new layout for a UISF. The general way to make a new layout is to use *makeLayout*, which takes layout information for first the horizontal dimension and then the vertical. A dimension can be either stretchy (with a minimum size in pixels but that will expand to fill the space it is given) or fixed (measured in pixels).

  The *setSize* function is a convenient function for setting the layout of a widget when both dimensions need to be fixed. It is defined as:

  $$\mathit{setSize}\ (w, h) = \mathit{setLayout}\ (\mathit{makeLayout}\ (\mathit{Fixed}\ w)\ (\mathit{Fixed}\ h))$$

  For example we can modify the previous example to both set a fixed layout for the overall widget, and attach titles to both the slider and display:

  $$
  \begin{array}{l}
  \mathit{ui}_1 :: \ \mathit{UISF}\ ()\ () \\
  \mathit{ui}_1 = \mathit{setSize}\ (150, 150)\ \$ \\
  \quad \textbf{proc}\ \_ \to \textbf{do} \\
  \qquad \mathit{ap} \leftarrow \mathit{title}\ \texttt{"Absolute Pitch"}\ (\mathit{hiSlider}\ 1\ (0, 100)\ 0) \prec ()\\
  \qquad \mathit{title}\ \texttt{"Pitch"}\ \mathit{display} \prec \mathit{pitch}\ \mathit{ap} \\
  \mathit{mui}_1 = \mathit{runMUI}'\ \mathit{ui}_1
  \end{array}
  $$

This MUI is shown in Figure 17.2(b).

- *pad* (w, n, e, s) *ui* adds w pixels of space to the "west" of the UISF *ui*, and n, e, and s pixels of space to the north, east, and south, respectively.

- The remaining four functions are used to control the relative layout of the widgets within a UISF. By default widgets are arranged top-to-bottom, but, for example, we could modify the previous UISF program to arrange the two widgets left-to-right:

```
ui2  :: UISF () ()
ui2  = leftRight $
  proc _ -> do
    ap <- title "Absolute Pitch" (hiSlider 1 (0,100) 0) -< ()
    title "Pitch" display -< pitch ap
mui2 = runMUI' ui2
```

  This MUI is shown in Figure 17.2(c).

Widget transformers can be nested (as demonstrated in some later examples), so a fair amount of flexibility is available.

17.3.3 MIDI Input and Output

An important application of events in Euterpea is real-time, interactive MIDI. There are two main UISF signal functions that handle MIDI, one for input and the other for output, but neither of them displays anything graphically:

```
midiIn  :: UISF (Maybe InputDeviceID)
                (SEvent [MidiMessage])
midiOut :: UISF (Maybe OutputDeviceID,
                 SEvent [MidiMessage]) ()
```

Except for the input and output deviceIDs (about which more will be said shortly), these signal functions are fairly straightforward: *midiOut* takes a stream of *MidiMessage* events and sends them to the MIDI output device (thus a signal sink), whereas *midiIn* generates a stream of *MidiMessage* events corresponding to the messages sent by the MIDI input device (thus a signal source)[3]. In both cases, note that the events carry *lists* of MIDI

[3]Technically, this is not a proper signal source because it accepts an input stream of *Maybe InputDeviceID*, but the way in which it generates MIDI messages makes it feel

messages, thus accounting for the possibility of simultaneous events.

The *MidiMessage* data type is defined as:

$$
\begin{array}{l}
\textbf{data}\ \mathit{MidiMessage} = \mathit{ANote}\ \{\mathit{channel} :: \mathit{Channel}, \mathit{key} :: \mathit{Key}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{velocity} :: \mathit{Velocity}, \mathit{duration} :: \mathit{Time}\} \\
\qquad\qquad\qquad\quad \mid\ \mathit{Std}\ \mathit{Message} \\
\quad \textbf{deriving}\ \mathit{Show}
\end{array}
$$

A *MidiMessage* is either an *ANote*, which allows us to specify a note with duration, or is a standard MIDI *Message*. MIDI does not have a notion of duration, but rather has separate *NoteOn* and *NoteOff* messages. With *ANote*, the design above is a bit more convenient, although what happens "behind the scenes" is that each *ANote* is transformed into a *NoteOn* and *NoteOff* event.

The *Message* data type is described in Chapter 14, and is defined in the *Codec.Midi* module. Its most important functionality is summarized here:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Message} = \\
\quad\ \ \text{-- Channel Messages} \\
\quad\ \ \mathit{NoteOff}\qquad\quad\ \{\mathit{channel} :: \mathit{Channel}, \mathit{key} :: \mathit{Key}, \mathit{velocity} :: \mathit{Velocity}\} \\
\quad \mid \mathit{NoteOn}\qquad\quad\ \ \{\mathit{channel} :: \mathit{Channel}, \mathit{key} :: \mathit{Key}, \mathit{velocity} :: \mathit{Velocity}\} \\
\quad \mid \mathit{ProgramChange}\ \{\mathit{channel} :: \mathit{Channel}, \mathit{preset} :: \mathit{Preset}\} \\
\quad \mid \ldots \\
\quad\ \ \text{-- Meta Messages} \\
\quad \mid \mathit{TempoChange}\ \mathit{Tempo} \mid \\
\quad \mid \ldots \\
\textbf{deriving}\ (\mathit{Show}, \mathit{Eq})
\end{array}
$$

MIDI's notion of a "key" is the key pressed on a MIDI instrument, not to be confused with "key" as in "key signature." Also, MIDI's notion of "velocity" is the rate at which the key is pressed, and is roughly equivalent to what we have been calling "volume." So, for example, a MIDI message *NoteOn* c k v plays MIDI key k on MIDI channel c with velocity v.

17.3.4 MIDI Device IDs

Before we can create an example using *midiIn* or *midiOut*, we first must consider their other arguments: *InputDeviceID* and *OutputDeviceID*. 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

---

very much like a source.

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. Thus, the only way to get an input or output device ID is by selection with one of the following widgets:

$$\begin{aligned}&\mathit{selectInput} \;\; :: \mathit{UISF}\;()\;(\mathit{Maybe}\;\mathit{InputDeviceID})\\&\mathit{selectOutput} :: \mathit{UISF}\;()\;(\mathit{Maybe}\;\mathit{OutputDeviceID})\end{aligned}$$

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, allowing the user to select one of them. In the event that there are no available devices, the widget can then return *Nothing*.

With these functions, we can now create an example using MIDI output. Let's modify our previous MUI program to output an *ANote* message every time the absolute pitch changes:

```
ui3 :: UISF () ()
ui3 = proc _ -> do
  devid <- selectOutput -< ()
  ap <- title "Absolute Pitch" (hiSlider 1 (0,100) 0) -< ()
  title "Pitch" display -< pitch ap
  uap <- unique -< ap
  midiOut -< (devid, fmap (\k -> [ANote 0 k 100 0.1]) uap)
mui3 = runMUI' ui3
```

The *unique* signal function used here is an example of a *mediator*, or a signal function that mediates between continuous and discrete signals. We will explore more mediators in Section 17.4.1, but in this case, note that *unique* will generate an event whenever its input, the continuous absolute pitch stream, changes. Each of those events, named *uap* above, carries the new absolute pitch, and that pitch is used directly as the MIDI key field in *ANote*.

To understand how that last part is done on the *midiOut* line, recall that *fmap* is the primary method in the *Functor* class as described in Section 7.4.3, and the *Maybe* type is an instance of *Functor*. Therefore, since *SEvent* is a type synonym for *Maybe*, the use of *fmap* above is valid—and all it does is apply the functional argument to the value "attached to" the event, which in this case is an absolute pitch.

For an example using MIDI input as well, here is a simple program that copies each MIDI message verbatim from the selected input device to the selected output device:

$$
\begin{array}{l}
ui_4 \;\; :: UISF\ ()\ () \\
ui_4 \;\; = \mathbf{proc}\ \_ \rightarrow \mathbf{do} \\
\quad mi \leftarrow selectInput \ \prec ()\\
\quad mo \leftarrow selectOutput \prec ()\\
\quad m \;\; \leftarrow midiIn \quad\;\; \prec mi\\
\quad midiOut \prec (mo, m)\\
mui_4 = runMUI'\ ui_4
\end{array}
$$

Since determining device IDs for both input and output is common, we define a simple signal function to do both:

$$
\begin{array}{l}
getDeviceIDs = topDown\ \$\\
\quad \mathbf{proc}\ () \rightarrow \mathbf{do}\\
\qquad mi \;\; \leftarrow selectInput \ \prec ()\\
\qquad mo \;\; \leftarrow selectOutput \prec ()\\
\qquad outA \prec (mi, mo)
\end{array}
$$

17.3.5 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 *UISF* value into a *IO* value—i.e. the UISF needs to be "run." We can do this using one of the following two functions, the first of which we have already been using:

$$
\begin{array}{l}
runMUI' :: \qquad\qquad\quad UISF\ ()\ () \rightarrow IO\ ()\\
runMUI \;\, :: UIParams \rightarrow UISF\ ()\ () \rightarrow IO\ ()
\end{array}
$$

Executing *runMUI′ ui* or *runMUI params ui* will create a single MUI window whose behavior is governed by the argument *ui* :: *UISF* () (). The additional *UIParams* argument of *runMUI* contains parameters that can affect the appearance and performance of the MUI window that is created. There is a default value of *UIParams* that is typical for regular MUI usage, and *runMUI′* is defined using it:

$$
\begin{array}{l}
defaultMUIParams :: UIParams\\
runMUI' = runMUI\ defaultMUIParams
\end{array}
$$

When using *runMUI*, it is advisable to simply modify the default value rather than building a whole new *UIParams* value. The easiest way to do this is with Haskell's *record syntax*.

There are many fields of data in a value of type *UIParams*, but we will focus only on the *uiTitle* and *uiSize*, which will control the value displayed in the title bar of the graphical window and the initial size of the window respectively. Thus, the title is a *String* value and the size is a *Dimension* value (where *Dimension* is a type synonym for (Int, Int), which in turn represents a width and height measured in pixels). By default, the size is $(300, 300)$ and the title is `"MUI"`, but we can change these like so:

$$
\begin{aligned}
&mui'_4 = runMUI\ (defaultMUIParams \\
&\qquad\qquad\qquad\qquad \{uiTitle = \texttt{"MIDI Input / Output UI"}, \\
&\qquad\qquad\qquad\qquad\ \ uiSize\ \ = (200, 200)\}) \\
&\qquad\qquad\qquad ui_4
\end{aligned}
$$

This version of mui_4 (from the previous subsection) will run identically to the original except for the fact that its title will read "MIDI Input / Output UI" and its initial size will be smaller.

17.4 Non-Widget Signal Functions

All of the signal functions we have seen so far are effectful widgets. That is, they all do something graphical or audible when they are used. For regular computation, we have been using pure functions (which we can insert arbitrarily in arrow syntax or lift with *arr* otherwise). However, there are signal functions that are important and useful which have no visible effects. We will look at a few different types of these signal functions in this section.

Details: Note that the mediators and folds in the next two subsections are generic signal functions, and are not restricted to use only in MUIs. To highlight this, we present them with the *SF* type rather than the *UISF* type. However, they can be (and often are) used as *UISF*s in MUIs.

The timers and delay functions in Subsections 17.4.3 and 17.4.4 require the MUI's internal notion of time, and so we present those directly with the *UISF* type.

17.4.1 Mediators

In order to use event streams in the context of continuous signals, Euterpea defines a set of functions that mediate between the continuous and the discrete. These "mediators," as well as some functions that deal exclusively with events, are shown in Figure 17.4 along with their type signatures and brief descriptions. Their use will be better understood through some examples that follow in Section 17.5.

$unique :: Eq\ a \Rightarrow SF\ a\ (SEvent\ a)$
    -- Generates an event whenever the input changes
$edge :: SF\ Bool\ (SEvent\ ())$
    -- Generates an event whenever the input changes from *False* to *True*
$hold :: a \rightarrow SF\ (SEvent\ a)\ a$
    -- *hold* x begins as value x, but changes to the subsequent values
    -- attached to each of its input events
$accum :: a \rightarrow SF\ (SEvent\ (a \rightarrow a))\ a$
    -- *accum* x starts with the value x, but then applies the function
    -- attached to the first event to x to get the next value, and so on
$now :: SF\ ()\ (SEvent\ ())$
    -- Creates a single event "now" and forever after does nothing.
$evMap :: SF\ a\ b \rightarrow SF\ (SEvent\ a)\ (SEvent\ b)$
    -- Lifts a continuous signal function into one that handles events
$mergeE :: (a \rightarrow a \rightarrow a) \rightarrow SEvent\ a \rightarrow SEvent\ a \rightarrow SEvent\ a$
    -- *mergeE* $f\ e_1\ e_2$ merges two events, using f to resolve two *Just* values
$(\sim\!\!+\!\!\!+) :: SEvent\ [a] \rightarrow SEvent\ [a] \rightarrow SEvent\ [a]$
    -- An infix specialization of *mergeE* to lists
$(\sim\!\!+\!\!\!+) = mergeE\ (+\!\!\!+)$

Figure 17.4: Mediators Between the Continuous and the Discrete

17.4.2 Folds

In traditional functional programming, a folding, or reducing, operation is one that joins together a set of data. The typical case would be an operation that operates over a list of data, such as a function that sums all elements of a list of numbers.

There are a few different ways given in Euterpea to fold together signal functions to create new ones:

$$
\begin{aligned}
&\mathit{maybeA} :: \mathit{SF}\ ()\ c \rightarrow \mathit{SF}\ b\ c \rightarrow \mathit{SF}\ (\mathit{Maybe}\ b)\ c \\
&\mathit{concatA} :: [\mathit{SF}\ b\ c] \rightarrow \mathit{SF}\ [b]\ [c] \\
&\mathit{runDynamic} :: \mathit{SF}\ b\ c \rightarrow \mathit{SF}\ [b]\ [c]
\end{aligned}
$$

- *maybeA* is a fold over the *Maybe* (or *SEvent*) data type. The signal function *maybeA* n j accepts as input a stream of *Maybe* b values; at any given moment, if those values are *Nothing*, then the signal function behaves like n, and if they are *Just* b, then it behaves like j.

- The *concatA* fold takes a list of signal functions and converts them to a single signal function whose streaming values are themselves lists. For example, perhaps we want to display a bunch of buttons to a user in a MUI window. Rather than coding them in one at a time, we can use *concatA* to fold them into one operation that will return their results altogether in a list. In essence, we are *concat*enating the signal functions together.

- The *runDynamic* signal function is similar to *concatA* except that it takes a single signal function as an argument rather than a list. What, then, does it fold over? Instead of folding over the static signal function list, it folds over the $[b]$ list that it accepts as its input streaming argument.

The *concatA* and *runDynamic* signal functions are definitely similar, but they are also subtly different. With *concatA*, there can be many different signal functions that are grouped together, but with *runDynamic*, there is only one. However, *runDynamic* may have a variable number of internally running signal functions at runtime because that number depends on a streaming argument. *concatA* is fixed once it is created.

17.4.3 Timers

The Euterpea MUI has an implicit notion of elapsed time, but it can be made explicit by the following signal source:

$$getTime :: UISF\ ()\ Time$$

where *Time* is a type synonym for *Double*.

Although the explicit time may be desired, some MUI widgets depend on the time implicitly. For example, the following signal function creates a *timer*:

$$timer :: UISF\ DeltaT\ (SEvent\ ())$$

In practice, $timer \multimap i$ takes a signal i that represents the timer interval (in seconds), and generates an event stream, where each pair of consecutive events is separated by the timer interval. Note that the timer interval is itself a signal, so the timer output can have varying frequency.

To see how a timer might be used, let's modify our MUI working example from earlier so that, instead of playing a note every time the absolute pitch changes, we will output a note continuously, at a rate controlled by a second slider:

```
ui5 :: UISF () ()
ui5 = proc _ -> do
  devid <- selectOutput -< ()
  ap    <- title "Absolute Pitch" (hiSlider 1 (0,100) 0) -< ()
  title "Pitch" display -< pitch ap
  f     <- title "Tempo" (hSlider (1,10) 1) -< ()
  tick  <- timer -< 1/f
  midiOut -< (devid, fmap (const [ANote 0 ap 100 0.1]) tick)
  -- Pitch Player with Timer
mui5 = runMUI' ui5
```

Note that the rate of *ticks* is controlled by the second slider—a larger slider value causes a smaller time between ticks, and thus a higher frequency, or tempo.

In some cases, the simple unit events of the *timer* are not enough. Rather, we would like each event to be different while we progress through a predetermined sequence. To do this, we can use the *genEvents* signal function:

$$genEvents :: [b] \rightarrow UISF\ DeltaT\ (SEvent\ b)$$

Just like *timer*, this signal function will output events at a variable frequency, but each successive event will contain the next value in the given list. When every value of the list *lst* has been emitted, *genEvents lst* will never again produce an event.

17.4.4 Delays

Another way in which a widget can use time implicitly is in a *delay*. Euterpea comes with five different delaying widgets, which each serve a specific role depending on whether the streams are continuous or event-based and if the delay is a fixed length or can be variable:

$$\begin{aligned}
&\mathit{delay} &&:: b \rightarrow \mathit{UISF}\ b\ b\\
&\mathit{fcdelay} &&:: b \rightarrow \mathit{DeltaT} \rightarrow \mathit{UISF}\ b\ b\\
&\mathit{fdelay} &&:: \mathit{DeltaT} \rightarrow \mathit{UISF}\ (\mathit{SEvent}\ b)\ (\mathit{SEvent}\ b)\\
&\mathit{vdelay} &&:: \mathit{UISF}\ (\mathit{DeltaT}, \mathit{SEvent}\ b)\ (\mathit{SEvent}\ b)\\
&\mathit{vcdelay} &&:: \mathit{DeltaT} \rightarrow b \rightarrow \mathit{UISF}\ (\mathit{DeltaT}, b)\ b
\end{aligned}$$

To start, we will examine the most straightforward one. The *delay* function creates what is called a "unit delay", which can be thought of as a delay by the shortest amount of time possible. This delay should be treated in the same way that one may treat a δt in calculus; that is, although one can assume that a delay takes place, the amount of time delayed approaches zero. Thus, in practice, this should be used only in continuous cases and should only be used as a means to initialize arrow feedback.

The rest of the delay operators delay by some amount of actual time, and we will look at each in turn. *fcdelay b t* will emit the constant value b for the first t seconds of the output stream and will from then on emit its input stream delayed by t seconds. The name comes from "fixed continuous delay."

One potential problem with *fcdelay* is that it makes no guarantees that every instantaneous value on the input stream will be seen in the output stream. This should not be a problem for continuous signals, but for an event stream, it could mean that entire events are accidentally skipped over. Therefore, there is a specialized delay for event streams: *fdelay t* guarantees that every input event will be emitted, but in order to achieve this, it is not as strict about timing—that is, some events may end up being over delayed.

Due to the nature of events, we no longer need an initial value for output: for the first t second, there will simply be no events emitted.

We can make both of the above delay widgets a little more complicated by introducing the idea of a variable delay. For instance, we can expand the capabilities of *fdelay* into *vdelay*. Now, the delay time is part of the signal, and it can change dynamically. Regardless, this event-based version will still guarantee that every input event will be emitted. "*vdelay*" can be read "variable delay."

For the variable continuous version, we must add one extra input parameter to prevent a possible space leak. Thus, the first argument to *vcdelay* is the maximum amount that the widget can delay. Due to the variable nature of *vcdelay*, some portions of the input signal may be omitted entirely from the output signal while others may even be outputted more than once. Thus, once again, it is highly advised to use *vdelay* rather than *vcdelay* when dealing with event-based signals.

17.5 Musical Examples

In this section we work through three larger musical examples that use Euterpea's MUI in interesting ways.

17.5.1 Chord Builder

This MUI will display a collection of chord types (Maj, Maj7, Maj9, min, min7, min9, and so on), one of which is selectable via a radio button. Then when a key is pressed on a MIDI keyboard, the selected chord is built and played using that key as the root.

To begin, we define a "database" that associates chord types with their intervals starting with the root note:

```
chordIntervals :: [(String, [Int])]
chordIntervals = [("Maj",   [4,3,5]),   ("Maj7",   [4,3,4,1]),
                  ("Maj9",  [4,3,4,3]), ("Maj6",   [4,3,2,3]),
                  ("min",   [3,4,5]),   ("min7",   [3,4,3,2]),
                  ("min9",  [3,4,3,4]), ("min7b5", [3,3,4,2]),
                  ("mMaj7", [3,4,4,1]), ("dim",    [3,3,3]),
                  ("dim7",  [3,3,3,3]), ("Dom7",   [4,3,3,2]),
                  ("Dom9",  [4,3,3,4]), ("Dom7b9", [4,3,3,3])]
```

We will display the list of chords on the screen as radio buttons for the user to click on.

The *toChord* function takes an input MIDI message as the root note, and the index of the selected chord, and outputs the notes of the selected chord.

```
toChord :: Int -> MidiMessage -> [MidiMessage]
toChord i m =
  case m of
    Std (NoteOn c k v)    -> f NoteOn c k v
    Std (NoteOff c k v)   -> f NoteOff c k v
    _ -> []
  where f g c k v = map (\k' -> Std (g c k' v))
                        (scanl (+) k (snd (chordIntervals !! i)))
```

Details: $scanl :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow [a]$ is a standard Haskell function that is like $foldl :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$, except that every intermediate result is returned, collected together in a list.

The overall MUI is laid out in the following way: On the left side, the list of input and output devices are displayed top-down. On the right is the list of chord types. We take the name of each chord type from the *chordIntervals* list to create the radio buttons.

When a MIDI input event occurs, the input message and the currently selected index to the list of chords is sent to the *toChord* function, and the resulting chord is then sent to the Midi output device.

```
buildChord :: UISF () ()
buildChord = leftRight $
  proc _ -> do
    (mi, mo) <- getDeviceIDs -< ()
    m        <- midiIn -< mi
    i        <- topDown $ title "Chord Type" $
                  radio (fst (unzip chordIntervals)) 0 -< ()
    midiOut -< (mo, fmap (concatMap $ toChord i) m)
chordBuilder = runMUI (defaultMUIParams
                          {uiTitle = "Chord Builder",
                           uiSize  = (600, 400)})
```

$$buildChord$$

Details: $unzip :: [(a, b)] \to ([a], [b])$ is a standard Haskell function that does the opposite of $zip :: [a] \to [b] \to [(a, b)]$.

$concatMap :: (a \to [b]) \to [a] \to [b]$ is another standard Haskell function that acts as a combination of map and $concat$. It maps the given function over the given list and then concatenates all of the outputs into a single output list.

Figure 17.5 shows this MUI in action.

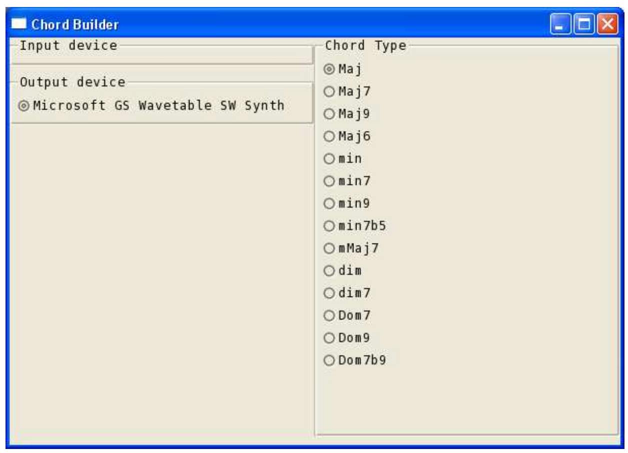


Figure 17.5: A Chord Builder MUI

17.5.2 Chaotic Composition

In this section we describe a UISF that borrows some ideas from Gary Lee Nelson's composition "Bifurcate Me, Baby!" [Nel95].

The basic idea is to evaluate a formula called the *logistic growth function*, from a branch of mathematics called chaos theory, at different points and convert the values to musical notes. The growth function is given by the recurrence equation:

$$x_{n+1} = rx_n(1 - x_n)$$

Mathematically, we start with an initial population x_0 and iteratively apply the growth function to it, where r is the growth rate. For certain values of r, the population stabilizes to a certain value, but as r increases, the period doubles, quadruples, and eventually leads to chaos. It is one of the classic examples of chaotic behavior.

We can capture the growth rate equation above in Haskell by defining a function that, given a rate r and current population x, generates the next population:

```
grow :: Double → Double → Double
grow r x = r ∗ x ∗ (1 − x)
```

To generate a time-varying population, the *accum* signal function comes in handy. *accum* is one of the mediators mentioned in Section 17.4.1: it takes an initial value and an event signal carrying a modifying function, and it updates the current value by applying the function to it.

```
 ...
 r   ← title "Growth rate" $ withDisplay (hSlider (2.4, 4.0) 2.4) −≺ ()
pop ← accum 0.1 −≺ fmap (const (grow r)) tick
 ...
```

The *tick* above is the "clock tick" that drives the simulation. We wish to define a signal *tick* that pulsates at a given frequency specified by a slider.

```
 ...
 f    ← title "Frequency" $ withDisplay (hSlider (1, 10) 1) −≺ ()
tick ← timer −≺ 1/f
 ...
```

We also need a simple function that maps a population value to a musical note. As usual, this can be done in a variety of ways—here is one way:

```
popToNote :: Double → [MidiMessage]
popToNote x = [ANote 0 n 64 0.05]
                 where n = truncate (x ∗ 127)
```

Finally, to play the note at every tick, we simply apply *popToNote* to every value in the time-varying population *pop*. *fmap* makes this straightforward. Putting it all together, we arrive at:

```
bifurcateUI :: UISF () ()
bifurcateUI = proc _ → do
  mo  ← selectOutput −≺ ()
  f   ← title "Frequency" $ withDisplay (hSlider (1, 10) 1) −≺ ()
```

```
    tick ← timer ⤙ 1/f
    r    ← title "Growth rate" $ withDisplay (hSlider (2.4, 4.0) 2.4) ⤙ ()
    pop  ← accum 0.1 ⤙ fmap (const (grow r)) tick
    _    ← title "Population" $ display ⤙ pop
    midiOut ⤙ (mo, fmap (const (popToNote pop)) tick)
  bifurcate = runMUI (defaultMUIParams
                          {uiTitle = "Bifurcate!",
                           uiSize  = (300, 500)})
                        bifurcateUI
```

17.5.3 MIDI Echo Effect

As a final example we present a program that receives a MIDI event stream and, in addition to playing each note received from the input device, it also echoes the note at a given rate, while playing each successive note more softly until the velocity reduces to 0.

The key component we need for this problem is a delay function that can delay a given event signal for a certain amount of time. Recall that the function *vdelay* takes a time signal, the amount of time to delay, and an input signal, and returns a delayed version of the input signal.

There are two signals we want to attenuate, or "decay." One is the signal coming from the input device, and the other is the delayed and decayed signal containing the echoes. In the code shown below, they are denoted as m and s, respectively. First we merge the two event streams into one, and then remove events with empty MIDI messages by replacing them with Nothing. The resulting signal m' is then processed further as follows.

The MIDI messages and the current decay rate are processed with *decay*, which softens each note in the list of messages. Specifically, *decay* works by reducing the velocity of each note by the given rate and removing the note if the velocity drops to 0. The resulting signal is then delayed by the amount of time determined by another slider f, producing signal s. Signal s is then merged with m in order to define m' (recall from Section17.4.1 that ~++ merges to event lists), thus closing the loop of the recursive signal. Finally, m' is sent to the output device.

```
echoUI :: UISF () ()
echoUI = proc _ → do
  (mi, mo) ← getDeviceIDs ⤙ ()
  m ← midiIn ⤙ mi
```

$r \leftarrow title$ `"Decay rate"` $\$\ withDisplay\ (hSlider\ (0, 0.9)\ 0.5) \prec ()$
$f \leftarrow title$ `"Echoing frequency"` $\$\ withDisplay\ (hSlider\ (1, 10)\ 10) \prec ()$
$\mathbf{rec}\ s \leftarrow vdelay \prec (1/f, fmap\ (mapMaybe\ (decay\ 0.1\ r))\ m')$
$\quad \mathbf{let}\ m' = m \sim\!\!+\!\!+\ s$
$midiOut \prec (mo, m')$
$echo = runMUI'\ echoUI$

$$
\begin{aligned}
&decay :: Time \to Double \to MidiMessage \to Maybe\ MidiMessage\\
&decay\ dur\ r\ m =\\
&\quad \mathbf{let}\ f\ c\ k\ v\ d = \mathbf{if}\ v > 0\\
&\qquad\qquad\qquad\quad \mathbf{then\ let}\ v' = truncate\ (fromIntegral\ v * r)\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ Just\ (ANote\ c\ k\ v'\ d)\\
&\qquad\qquad\qquad\quad \mathbf{else}\ \ Nothing\\
&\quad \mathbf{in\ case}\ m\ \mathbf{of}\\
&\qquad ANote\ c\ k\ v\ d \qquad\quad \to f\ c\ k\ v\ d\\
&\qquad Std\ (NoteOn\ c\ k\ v) \to f\ c\ k\ v\ dur\\
&\qquad \_ \qquad\qquad\qquad\quad\ \ \to Nothing
\end{aligned}
$$

17.6 Special Purpose and Custom Widgets

Although the widgets and signal functions described so far enable the creation of many basic MUIs, there are times when something more specific is required. Thus, in this section, we will look at some special purpose widgets as well as some functions that aid in the creation of custom widgets.

Some of the functions described in this subsection are included in Euterpea by default, but others require extra imports of specific Euterpea modules. We will note this where applicable.

17.6.1 Realtime graphs, histograms

So far, the only way to display the value of a stream in the MUI is to use the *display* widget. Although this is often enough, there may be times when another view is more enlightening. For instance, if the stream represents a sound wave, then rather than displaying the instantaneous values of the wave as numbers, we may wish to see them graphed.

Euterpea provides support for a few different widgets that will graph streaming data visually.

$realtimeGraph \quad :: RealFrac\ a \Rightarrow Layout$

$$
\begin{aligned}
& \qquad\qquad \rightarrow \mathit{Time} \rightarrow \mathit{Color} \rightarrow \mathit{UISF}\ [(a, \mathit{Time})]\ () \\
& \mathit{histogram} \qquad\quad :: \mathit{RealFrac}\ a \Rightarrow \mathit{Layout} \\
& \qquad\qquad \rightarrow \mathit{UISF}\ (\mathit{SEvent}\ [a])\ () \\
& \mathit{histogramWithScale} :: \mathit{RealFrac}\ a \Rightarrow \mathit{Layout} \\
& \qquad\qquad \rightarrow \mathit{UISF}\ (\mathit{SEvent}\ [(a, \mathit{String})])\ ()
\end{aligned}
$$

Note that each of these three functions requires a *Layout* argument (recall the *Layout* data type from Section 17.3.2); this is because the layout of a graph is not as easily inferred as that for, say, a button.

We will walk through the descriptions of each of these widgets:

- *realtimeGraph* l t c will produce a graph widget with layout l. This graph will accept as input a stream of events of pairs of values and time[4]. The values are plotted vertically in color c, and the horizontal axis represents time, where the width of the graph represents an amount of time t.

- The histogram widgets' input are events that each contain a complete set of data. The data are plotted as a histogram within the given layout. For the histogram with the scale, each value must be paired with a *String* representing its label, and the labels are printed under the plot.

These widgets will prove useful when we are dealing with sound signals directly in future chapters.

17.6.2 More MIDI Widgets

In Sections 17.3.3 and 17.3.4, we presented simple widgets for selecting devices and polling and playing midi messages. However, these widgets allow for only one input device and one output device at a time. For a more complex scenario where multiple devices are to be used simultaneously, we have the following four widgets:

$$
\begin{aligned}
& \mathit{midiInM} && :: \mathit{UISF}\ [\mathit{InputDeviceID}]\ (\mathit{SEvent}\ [\mathit{MidiMessage}]) \\
& \mathit{midiOutM} && :: \mathit{UISF}\ [(\mathit{OutputDeviceID}, \mathit{SEvent}\ [\mathit{MidiMessage}])]\ () \\
& \mathit{selectInputM} && :: \mathit{UISF}\ ()\ [\mathit{InputDeviceID}] \\
& \mathit{selectOutputM} && :: \mathit{UISF}\ ()\ [\mathit{OutputDeviceID}]
\end{aligned}
$$

[4] These events are represented as a list rather than using the *SEvent* type because there may be more than one event at the same time. The absence of any events would be indicated by an empty list.

The M on the end can be read as "Multiple." These widgets can be used just like their singular counterparts to handle MIDI, except that they allow for multiple simultaneous device usage.

We can add even more behavior into the midi output widgets by considering a *buffered* output. When using *midiOut* (or *midiOutM*), all of the MIDI messages sent to the device are immediately played, but sometimes, we would prefer to queue messages up for playback later. We can do this with the following two midi output widgets:

```
midiOutB  :: UISF (Maybe OutputDeviceID,
                   BufferOperation MidiMessage) Bool
midiOutMB :: UISF [(OutputDeviceID,
                   BufferOperation MidiMessage)] Bool
```

Notice that these two widgets have a *Bool* output stream; this stream is *True* when the buffer is empty and there is nothing queued up to play and *False* otherwise. The *BufferOperation* data type gives information along with the MIDI messages about when or how to play the messages. It is defined as follows:

```
data BufferOperation b =
    NoBOp
  | ClearBuffer
  | SkipAheadInBuffer DeltaT
  | MergeInBuffer [(DeltaT, b)]
  | AppendToBuffer [(DeltaT, b)]
  | SetBufferPlayStatus Bool (BufferOperation b)
  | SetBufferTempo Tempo     (BufferOperation b)
```

where

- *NoBOp* indicates that there is no new information for the buffer.
- *ClearBuffer* erases the current buffer.
- *SkipAheadInBuffer t* skips ahead in the buffer by *t* seconds.
- *MergeInBuffer ms* merges messages *ms* into the buffer to play concurrently with what is currently playing.
- *AppendToBuffer ms* adds messages *ms* to the end of the buffer to play immediately following whatever is playing.
- *SetBufferPlayStatus p b* indicates whether the buffer should be playing (*True*) or paused (*False*).

- *SetBufferTempo t b* sets the play speed of the buffer to t (the default is 1, indicating realtime).

Note that the final two options recursively take a buffer operation, meaning that they can be attached to any other buffer operation as additional modifications.

Details: The *midiOutB* and *midiOutMB* widgets are essentially the regular *midiOut* widgets connected to an *eventBuffer*. The *eventBuffer* signal function can also be used directly to buffer any kind of data that fits into the *BufferOperation* format. It can be brought into scope by importing FRP.UISF.AuxFunctions.

In practice, the most common time to use the buffered midi output widgets as opposed to the regular ones is when dealing with *Music* values. Thus, Euterpea.IO.MUI.MidiWidgets also exports the following function:

$$\begin{aligned} &\mathit{musicToMsgs} :: \mathit{Maybe}\ [\mathit{InstrumentName}] \rightarrow \mathit{Music1} \\ &\qquad\qquad\qquad \rightarrow [(\mathit{DeltaT}, \mathit{MidiMessage})] \end{aligned}$$

The first argument should be a *Just* value if the *Music1* value is infinite and *Nothing* otherwise. If it is a *Just* value, then its value should be the override for the instrument channels.

The *musicToMsgs* function will convert a *Music1* value into a format that can be easily sent to the buffered midi widget. Once converted in this way, it can be wrapped by *MergeInBuffer* or *AppendToBuffer* to be sent to the buffer.

17.6.3 Virtual Instruments

Euterpea provides two special widgets that create virtual instruments that the user can interact with: a piano and a guitar. These two widgets are not fully supported by Euterpea at present, so to bring them into scope, we will need to import Euterpea.Experimental.

import *Euterpea.Experimental*

Details: Euterpea's Experimental package contains functions and features that are still in process. They may be unreliable, and they are likely to change in future versions of Euterpea. Thus, feel free to experiment with them, but use them with caution.

The piano and guitar are virtual instruments in the MUI that look and behave like a piano keyboard or guitar strings: the strings can be plucked or the piano keys pressed with either the mouse or keyboard, and the output is in the form of MIDI messages. Note that these widgets do not actually produce any sound. Thus, in most cases, the output should then be sent to a MIDI output widget.

The two widgets have the following similar types:

$$
\begin{aligned}
&\mathit{guitar} :: \mathit{GuitarKeyMap} \rightarrow \mathit{Midi.Channel}\\
&\qquad \rightarrow \mathit{UISF}\ (\mathit{InstrumentData}, \mathit{SEvent}\ [\mathit{MidiMessage}])\\
&\qquad\qquad (\mathit{SEvent}\ [\mathit{MidiMessage}])\\
&\mathit{piano} :: \mathit{PianoKeyMap} \rightarrow \mathit{Midi.Channel}\\
&\qquad \rightarrow \mathit{UISF}\ (\mathit{InstrumentData}, \mathit{SEvent}\ [\mathit{MidiMessage}])\\
&\qquad\qquad (\mathit{SEvent}\ [\mathit{MidiMessage}])
\end{aligned}
$$

Given a key mapping and a MIDI channel, the functions make the virtual instrument widgets. The widgets themselves accept an *InstrumentData* argument, which contains some settings for the instrument, and a stream of input MIDI messages, and they produce a stream of MIDI messages. They do not make any sound themselves—these widgets are purely visual.

Let's look at how these widgets work in a little more detail. First, the widgets take a key map, either a *GuitarKeyMap* or a *PianoKeyMap*. These maps indicate what keyboard keys one can use to play the instruments with a standard computer keyboard. These are customizable values, but we provide a couple for use with a qwerty keyboard:

- $\mathit{defaultMap}_1$ treats the characters from Q to U as one octave from C2 to B3. The black notes are predictably at 2, 3, 5, 6, and 7. Holding a shift key while pressing the same keys plays the notes one octave higher.

- $\mathit{defaultMap}_2$ is the same as $\mathit{defaultMap}_1$ except that it uses the bottom two rows of the keyboard, with Z through M as the white keys and S through J (but not F) as black keys. Once again, hold shift for the higher octave.

- $defaultMap_0$ is for a four octave keyboard and uses both maps in sequence.

For the guitar, we provide *sixString*, a mapping using the first six columns of keys (e.g. 1, Q, A, Z would be the first column) to represent the six strings of the guitar.

The next argument to making a virtual instrument widget is a MIDI channel. Because they can create MIDI messages from just a mouse click, these widgets need information about what MIDI channel the messages should use. The *Midi.Channel* type is brought in from *Codec.Midi*, and it is a type synonym for *Int*—really, any number from 0 to 127 is probably an okay candidate, but it depends on what channels your MIDI devices support.

The streaming input to the widgets includes both MIDI messages (which will visually "play" on the instrument) as well as a value of *InstrumentData*. By default, one should use the value *defaultInstrumentData*, but this can be modified with the following three widgets:

```
addNotation  :: UISF InstrumentData InstrumentData
addTranspose :: UISF InstrumentData InstrumentData
addPedal     :: UISF InstrumentData InstrumentData
```

Each one will create a checkbox or slider to allow for adding notation (visual text that indicates what keys are what on the instrument), transposition (the ability to raise or lower the notes by some amount) or pedal (only used for the piano).

Now that we have some idea of how the widgets work, let's create a sample MUI that uses them both.

```
gAndPUI :: UISF () ()
gAndPUI = proc _ -> do
  (mi, mo) <- getDeviceIDs -< ()
  m <- midiIn -< mi
  settings <- addNotation -< defaultInstrumentData
  outG <- guitar sixString 1     -< (settings, Nothing)
  outP <- piano defaultMap0 0 -< (settings, m)
  midiOut -< (mo, outG ~++ outP)
gAndP = runMUI (defaultMUIParams { uiSize = (1050, 700),
                                   uiTitle = "Guitar and Piano"})
                 gAndPUI
```

This MUI will provide a checkbox for whether it should display notation or not and then shows both virtual instruments. Any messages played on the input MIDI device will be shown and heard as if played on the virtual piano.

17.6.4 A Graphical Canvas

In addition to the standard musical widgets, the musical user interface provides support for arbitrary graphical output. It does this via the *canvas* widget, which allows the user to "paint" graphics right into the MUI:

$$\begin{aligned}&\mathit{canvas} :: \mathit{Dimension} \rightarrow \mathit{UISF}\ (\mathit{Event}\ \mathit{Graphic})\ ()\\ &\mathit{canvas}' :: \mathit{Layout} \rightarrow (a \rightarrow \mathit{Dimension} \rightarrow \mathit{Graphic}) \rightarrow \mathit{UISF}\ (\mathit{Event}\ a)\ ()\end{aligned}$$

The main *canvas* widget takes a fixed size and displays in the MUI the most recent *Graphic* it received. The *canvas′* function is a little more complex as it can handle a changing size: rather than a fixed dimension, it accepts a layout and a function that, when given the dimension (which is generated at runtime based on the window size), can produce the appropriate graphic.

In either case, the user is responsible for generating the graphic that should be shown by generating a value of type *Graphic*. However, Euterpea does not export *Graphic* constructors by default, so we will need to add the following import to our file:

import *FRP.UISF.SOE*

The name of this import, SOE, comes from the book The Haskell School of Expression, the predecessor to this text. Rather than go into detail about the various types of graphics one can create with this import, we will leave it to the reader to read this other text or to look at the documentation directly. Instead, we will only point out three functions as we will use them in our upcoming example:

$$\begin{aligned}&\mathit{polygon} &&:: [(\mathit{Int}, \mathit{Int})] \rightarrow \mathit{Graphic}\\ &\mathit{rgb} &&:: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{RGB}\\ &\mathit{withColor}' &&:: \mathit{RGB} \rightarrow \mathit{Graphic} \rightarrow \mathit{Graphic}\end{aligned}$$

The *polygon* function takes a list of points and constructs a polygon treating them as vertices, the *rgb* function produces an *RGB* color from red, green, and blue values, and *withColor′* applies the given *RGB* color to the given *Graphic*.

In the following example, we will create three sliders to control the red, green, and blue values, and then we will use these to create a simple color swatch out of the *canvas* widget.

```
colorSwatchUI :: UISF () ()
colorSwatchUI = setSize (300, 220) $ pad (4, 0, 4, 0) $ leftRight $
    proc _ → do
      r ← newColorSlider "R" ⤙ ()
      g ← newColorSlider "G" ⤙ ()
      b ← newColorSlider "B" ⤙ ()
      e ← unique ⤙ (r, g, b)
      let rect = withColor′ (rgb r g b) (box ((0, 0), d))
      pad (4, 8, 0, 0) $ canvas d ⤙ fmap (const rect) e
  where
    d = (170, 170)
    newColorSlider l = title l $ withDisplay $ viSlider 16 (0, 255) 0
    box ((x, y), (w, h)) =
      polygon [(x, y), (x + w, y), (x + w, y + h), (x, y + h)]
colorSwatch = runMUI′ colorSwatchUI
```

We use the *polygon* function to create a simple box, and then we color it with the data from the sliders. Whenever the color changes, we redraw the box by sending a new *Graphic* event to the *canvas* widget.

17.6.5 [Advanced] mkWidget

In some cases, even the *canvas* widget is not powerful enough, and we would like to create our own custom widget. For this, there is the *mkWidget* function. To bring this into scope, we must import UISF's widget module directly:

```
import FRP.UISF.Widget (mkWidget)
```

The type of *mkWidget* is as follows:

```
mkWidget :: s
         → Layout
         → (a → s → Rect → UIEvent → (b, s, DirtyBit))
         → (Rect → Bool → s → Graphic)
         → UISF a b
```

This widget building function takes arguments particularly desgined to make a realtime, interactive widget. The arguments work like so:

- The first argument is an initial state for the widget. The widget will be able to internally keep track of state, and the value that it should start with is given here.

- The second argument is the layout of the widget.
- The third argument is the computation that this layout performs. Given an instantaneous value of the streaming input, the current state, the rectangle describing the current allotted dimensions, and the current *UIEvent*[5], it should produce an output value, a new state, and a *DirtyBit*, which is a boolean value indicating whether the visual representation of the widget will change.
- The final argument is the drawing routine. Given the rectangle describing the current allotted dimensions for the widget (the same as given to the computation function), a boolean indicating whether this widget is in focus, and the state, it produces the graphic that this widget will appear as.

The specifics of *mkWidget* are beyond the scope of this text, and those interested in making their own widgets are encouraged to look at the documentation of the UISF package. However, as a demonstration of its use, here we will show the definition of *canvas* using *mkWidget*.

$$
\begin{array}{l}
canvas\ (w, h) = mkWidget\ nullGraphic\ layout\ process\ draw \\
\quad \textbf{where} \\
\quad\quad layout = makeLayout\ (Fixed\ w)\ (Fixed\ h) \\
\quad\quad draw\ ((x, y), (w, h))\ \_ = translateGraphic\ (x, y) \\
\quad\quad process\ (Just\ g)\ \_\ \_\ \_ = ((), g, True) \\
\quad\quad process\ Nothing\ g\ \_\ \_ = ((), g, False)
\end{array}
$$

17.7 Advanced Topics

In the final section of this chapter, we will explore some advanced topics related to the MUI.

17.7.1 Banana brackets

When dealing with layout, we have so far shown two ways to apply the various layout transformers (e.g. *topDown*, *leftRight*, etc.) to signal func-

[5]The *UIEvent* can contain information like mouse clicks or key presses. For complete documentation on *UIEvent*, look to the FRP.UISF documentation.

tions. One way involves using the transformer on the whole signal function by applying it on the first line like so:

$$... = \textit{leftRight}\ \$\ \textbf{proc}\ \_ \rightarrow \textbf{do}\ ...$$

The other option is to apply the transformation in-line for the signal function it should act upon:

$$\begin{array}{l} ... \\ \quad x \leftarrow \textit{topDown}\ \textit{mySF} \prec y \\ ... \end{array}$$

However, the situation is not so clear cut, and at times, we may want a sub-portion of our signal function to have a different layout flow than the rest.

For example, assume we have a signal function that should have four buttons. The second and third buttons should be left-right aligned, but vertically, they together should be between the first and second. One way we may try to write this is like so:

$$\begin{array}{l} \textit{ui}_6 = \textit{topDown}\ \$\ \textbf{proc}\ \_ \rightarrow \textbf{do} \\ \quad b_1 \leftarrow \textit{button}\ \texttt{"Button 1"} \prec () \\ \quad (b_2, b_3) \leftarrow \textit{leftRight}\ (\textbf{proc}\ \_ \rightarrow \textbf{do} \\ \qquad b_2 \leftarrow \textit{button}\ \texttt{"Button 2"} \prec () \\ \qquad b_3 \leftarrow \textit{button}\ \texttt{"Button 3"} \prec () \\ \qquad \textit{returnA} \prec (b_2, b_3)) \prec () \\ \quad b_4 \leftarrow \textit{button}\ \texttt{"Button 4"} \prec () \\ \quad \textit{display} \prec b_1 \vee b_2 \vee b_3 \vee b_4 \end{array}$$

This looks a little funny, especially because we have an extra arrow tail (the $\prec$ part) after the inner *returnA* on the sixth line, but it gets the job done.

However, what if we wanted to do something with the value b_1 within the inner **proc** part? In its current state, b_1 is not in scope in there. We can add it to the scope, but we would have to explicitly accept that value from the outer scope. It would look like so:

$$\begin{array}{l} \textit{ui}'_6 = \textit{topDown}\ \$\ \textbf{proc}\ \_ \rightarrow \textbf{do} \\ \quad b_1 \leftarrow \textit{button}\ \texttt{"Button 1"} \prec () \\ \quad (b_2, b_3) \leftarrow \textit{leftRight}\ (\textbf{proc}\ b_1 \rightarrow \textbf{do} \\ \qquad b_2 \leftarrow \textit{button}\ \texttt{"Button 2"} \prec () \\ \qquad \textit{display} \prec b_1 \\ \qquad b_3 \leftarrow \textit{button}\ \texttt{"Button 3"} \prec () \\ \qquad \textit{returnA} \prec (b_2, b_3)) \prec b_1 \\ \quad b_4 \leftarrow \textit{button}\ \texttt{"Button 4"} \prec () \end{array}$$

$\mathit{display} \prec b_1 \vee b_2 \vee b_3 \vee b_4$

This is getting hard to deal with! Fortunately, there is an arrow syntax feature to help us with this known as *banana brackets*.

Banana brackets are a component of the arrow syntax that allows one to apply a function to one or more arrow commands without losing the scope of the arrow syntax. To use, one writes in the form:

$(\!| \; f \; \mathit{cmd}_1 \; \mathit{cmd}_2 ... \; |\!)$

where f is a function on arrow commands and cmd_1, cmd_2, etc. are arrow commands.

> **Details:** An *arrow command* is the portion of arrow syntax that contains the arrow and the input but not the binding to output. Generally, this looks like $\mathit{sf} \prec x$, but if it starts with **do**, then it can be an entire arrow in itself (albeit, one that does not start with **proc** $\_ \rightarrow$).

Banana brackets preserve the original arrow scope, so we can rewrite our example to:

$ui''_6 = \mathbf{proc}\ () \rightarrow \mathbf{do}$
  $b_1 \leftarrow \mathit{button}$ `"Button 1"` $\prec ()$
  $(b_2, b_3) \leftarrow (\!| \; \mathit{leftRight} \; (\mathbf{do}$
    $b_2 \leftarrow \mathit{button}$ `"Button 2"` $\prec ()$
    $\mathit{display} \prec b_1$
    $b_3 \leftarrow \mathit{button}$ `"Button 3"` $\prec ()$
    $\mathit{returnA} \prec (b_2, b_3)) \; |\!)$
  $b_4 \leftarrow \mathit{button}$ `"Button 4"` $\prec ()$
  $\mathit{display} \prec b_1 \vee b_2 \vee b_3 \vee b_4$

Note that we no longer need the **proc** $\_ \rightarrow$ in the third line nor do we have an arrow tail on the seventh line. That said, banana brackets do have a limitation in that the variables used internally are not exposed outside; that is, we still need the seventh line to explicitly return b_2 and b_3 in order to bind them to the outer scope in the third line so that they are visible when displayed on the last line.

17.7.2 General I/O From Within a MUI

So far, through specific widgets, we have shown how to perform specific effects through the MUI: one can poll MIDI devices, send MIDI output, display graphics on the screen, and so on. However, the MUI is capable of arbitrary *IO* actions. In general, arbitrary *IO* actions can be dangerous, so the functions that allow them are relegated to Euterpea.Experimental, and they should be used with care.

The first arbitrary *IO* arrow to consider is:

$$\mathit{initialAIO} :: \mathit{IO}\ d \to (d \to \mathit{UISF}\ b\ c) \to \mathit{UISF}\ b\ c$$

This function allows an *IO* action to be performed upon MUI initialization, the result of which is used to finish constructing the widget. Thus, its name can be read as "initial Arrow IO."

In practice one might use *initialAIO* to do something like read the contents of a file to be used at runtime. For instance, if we had a file called "songData" that contained data we would like to use in the MUI, we could use the following function:

$$\begin{aligned}&\mathit{initialAIO}\ (\mathit{readFile}\ \texttt{"songData"})\\ &\qquad\qquad (\lambda x \to \mathit{now} \ggg \mathit{arr}\ (\mathit{fmap}\ \$\ \mathit{const}\ x))\\ &\quad :: \mathit{UISF}\ ()\ (\mathit{SEvent}\ \mathit{String})\end{aligned}$$

This function will read the file and then produce a single event containing the contents of the file when the MUI first starts.

Performing an initial action is simple and useful, but at times, we would like the freedom to perform actions mid-execution as well, and for that, we have the following six functions:

$$\begin{aligned}&\mathit{uisfSource} &&:: \mathit{IO}\ c &&\to \mathit{UISF}\ ()\ c\\ &\mathit{uisfSink} &&:: (b \to \mathit{IO}\ ()) &&\to \mathit{UISF}\ b\ ()\\ &\mathit{uisfPipe} &&:: (b \to \mathit{IO}\ c) &&\to \mathit{UISF}\ b\ c\\ &\mathit{uisfSourceE} &&:: \mathit{IO}\ c &&\to \mathit{UISF}\ (\mathit{SEvent}\ ())\ (\mathit{SEvent}\ c)\\ &\mathit{uisfSinkE} &&:: (b \to \mathit{IO}\ ()) &&\to \mathit{UISF}\ (\mathit{SEvent}\ b)\ \ (\mathit{SEvent}\ ())\\ &\mathit{uisfPipeE} &&:: (b \to \mathit{IO}\ c) &&\to \mathit{UISF}\ (\mathit{SEvent}\ b)\ \ (\mathit{SEvent}\ c)\end{aligned}$$

The first three of these are for continuous-type actions and the last three are for event-based actions. As an example of a continuous action, one could consider a stream of random numbers:

$$\mathit{uisfSource}\ \mathit{randomIO} :: \mathit{Random}\ r \Rightarrow \mathit{UISF}\ ()\ r$$

Most *IO* actions are better handled by the event-based functions. For instance, we could update our file reading widget from earlier so that it is capable of reading a dynamically named file, and it can perform more than one read at runtime:

$$uisfPipeE\ readFile :: UISF\ (SEvent\ FilePath)\ (SEvent\ String)$$

Whenever this signal function is given an event containing a file, it reads the file and returns an event containing the contents.

Details: This sort of arbitrary *IO* access that the functions from this subsection allow can have negative effects on a program ranging from unusual behavior to performance problems to crashing. Research has been done to handle these problems, and a promising solution using what are called *resource types* has been proposed [WCLH11, WCH12]. However, Euterpea does not implement resource types, so it is left to the programmer to be exceptionally careful to use these appropriately.

17.7.3 Asynchrony

Although we have discussed the MUI as being able to act both continuously and discretely (event-based) depending on the required circumstances, in actual fact, the system is entirely built in a discrete way. When run, the MUI does many calculations per second to create the illusion of continuity, and as long as this sample rate is high enough, the illusion persists without any problem.

However, there are two primary ways in which the illusion of continuity fails:

- Computations can be sensitive to the sampling rate itself such that a low enough rate will cause poor behavior.
- Computations can be sensitive to the variability of the sampling rate such that drastic differences in the rate can cause poor behavior.

These are two subtly different problems, and we will address both with subtly different forms of *asynchrony*.

The idea of using asynchrony is to allow these sensitive computations to run separately from the MUI process so that they are unaffected by the MUI's sampling rate and are allowed to set and use their own arbitrary rate. We achieve this with the following functions:

$$\begin{aligned}&asyncUISFE \circ toAutomaton :: NFData\ b \Rightarrow\\&\quad SF\ a\ b \rightarrow UISF\ (SEvent\ a)\ (SEvent\ b)\end{aligned}$$

$$\begin{array}{ll} \mathit{clockedSFToUISF} & :: (\mathit{NFData}\ b, \mathit{Clock}\ c) \Rightarrow \\ \quad \mathit{DeltaT} \rightarrow \mathit{SigFun}\ c\ a\ b \rightarrow \mathit{UISF}\ a\ [(b, \mathit{Time})] & \end{array}$$

Note that the *SF* and *SigFun* types will be discussed further in Chapter 19), but they are both arrows, and thus we can lift pure functions of type $a \rightarrow b$ to them with the *arr* function. These two functions are designed to address the two different sampling rate pitfalls we raised above.

- *asyncUISFE* ∘ *toAutomaton* is technically a composition of two functions, but in Euterpea, it would be rare to use them apart. Together, they are used to deal with the scenario where a computation takes a long time to compute (or perhaps blocks internally, delaying its completion). This slow computation may have deleterious effects on the MUI, causing it to become unresponsive and slow, so we allow it to run asynchronously. The computation is lifted into the discrete, event realm, and for each input event given to it, a corresponding output event will be created eventually. Of course, the output event will likely not be generated immediately, but it will be generated eventually, and the ordering of output events will match the ordering of input events.

- The *clockedSFToUISF* function can convert a signal function with a fixed, virtual clockrate to a realtime UISF. The first input parameter is a buffer size in seconds that indicates how far ahead of real time the signal function is allowed to get, but the goal is to allow it to run at a fixed clockrate as close to realtime as possible. Thus, the output stream is a list of pairs providing the output values along with the timestamp for when they were generated. This should contain the right number of samples to approach real time, but on slow computers or when the virtual clockrate is exceptionally high, it will lag behind. This can be checked and monitored by checking the length of the output list and the time associated with the final element of the list on each time step.

Rather than show an example here, we will wait until Chapter 20 once the *SigFun* type has been introduced. An example that uses *clockedSFToUISF* can be found at the end of the chapter in Figure 20.5

Exercise 17.1 Define a MUI that has a text box in which the user can type a pitch using the normal syntax $(C, 4)$, $(D, 5)$, etc., and a pushbutton labelled "Play" that, when pushed, will play the pitch appearing in the textbox.

Hint: use the Haskell function $\mathit{reads} :: \mathit{Read}\ a \Rightarrow \mathit{String} \rightarrow [(a, \mathit{String})]$ to parse the input.

Exercise 17.2 Modify the previous example so that it has *two* textboxes, and plays both notes simultaneously when the pushbutton is pressed.

Exercise 17.3 Modify the previous example so that, in place of the pushbutton, the pitches are played at a rate specified by a horizontal slider.

Exercise 17.4 Define a MUI for a pseudo-keyboard that has radio buttons to choose one of the 12 pitches in the conventional chromatic scale. Every time a new pitch is selected, that note is played.

Exercise 17.5 Modify the previous example so that an integral slider is used to specify the octave in which the pitch is played.

Exercise 17.6 Leon Gruenbaum describes a "Samchillian Tip Tip Tip Cheeepeeeee," a MIDI keyboard based on intervals rather than fixed pitches. Your job is to define a "Cheepie Samchillian" as a MUI that has the following features:

- A three-element radio button to choose between three scales: chromatic, major, and whole-tone.
- Nine pushbuttons, corresponding to intervals (within the selected scale) of 0, +1, +2, +3, +4, -1, -2, -3, and -4.

Chapter 18

Sound and Signals

In this chapter we study the fundamental nature of sound and its basic mathematical representation as a signal. We also discuss discrete digital representations of a signal, which form the basis of modern sound synthesis and audio processing.

18.1 The Nature of Sound

Before studying digital audio, it's important that we first know what *sound* is. In essence, sound is the rapid compression and relaxation of air, which travels as a *wave* through the air from the physical source of the sound to, ultimately, our ears. The physical source of the sound could be the vibration of our vocal chords (resulting in speech or singing), the vibration of a speaker cone, the vibration of a car engine, the vibration of a string in a piano or violin, the vibration of the reed in a saxophone or of the lips when playing a trumpet, or even the (brief and chaotic) vibrations that result when our hands come together as we clap. The "compression and relaxation" of the air (or of a coiled spring) is called a *longitudinal* wave, in which the vibrations occur parallel to the direction of travel of the wave. In contrast, a rope that is fixed at one end and being shaken at the other, and a wave in the ocean, are examples of a *transverse* wave, in which the rope's and water's movement is perpendicular to the direction the wave is traveling.

[Note: There are some great animations of these two kinds of waves at: `http://www.computermusicresource.com/what.is.sound.html`.]

If the rate and amplitude of the sound are within a suitable range, we can *hear* the sound—i.e. it is *audible sound*. “Hearing” results when the vibrating air waves cause our ear drum to vibrate, in turn stimulating nerves that enter our brain. Sound above our hearing range (i.e. vibration that is too quick to induce any nerve impulses) is called *ultrasonic sound*, and sound below our hearing range is said to be *infrasonic*.

Staying within the analog world, sound can also be turned into an *electrical* signal using a *microphone* (or “mic” for short). Several common kinds of microphones are:

1. Carbon microphone. Based on the resistance of a pocket of carbon particles that are compressed and relaxed by the sound waves hitting a diaphram.
2. Condenser microphone. Based on the capacitance between two diaphrams, one being vibrated by the sound.
3. Dynamic microphone. Based on the inductance of a coil of wire suspended in a magnetic field (the inverse of a speaker).
4. Piezoelectric microphone. Based on the property of certain crystals to induce current when they are bent.

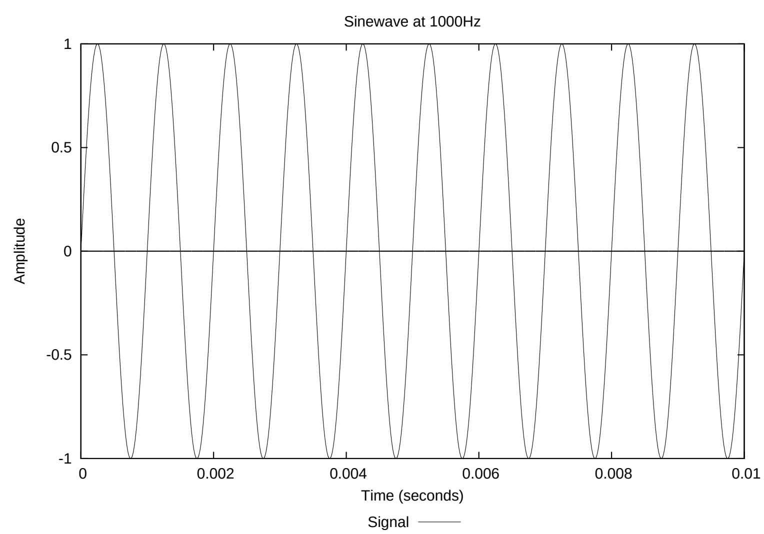


Figure 18.1: A Sine Wave

Perhaps the most common and natural way to represent a wave diagrammatically, whether it be a sound wave or electrical wave, longitudinal or transverse, is as a *graph* of its amplitude vs. time. For example, Figure 18.1 shows a *sinusiodal wave* of 1000 cycles per second, with an amplitude that varies beween +1 and -1. A sinusoidal wave follows precisely the definition of the mathematical sine function, but also relates strongly, as we shall soon see, to the vibration of sound produced by most musical instruments. In the remainder of this text, we will refer to a sinusoidal wave simply as a sine wave.

Acoustics is the study of the properties, in particular the propagation and reflection, of sound. *Psychoacoustics* is the study of the mind's interpretation of sound, which is not always as tidy as the physical properties that are manifest in acoustics. Obviously both of these are important areas of study for music in general, and therefore play an important role in generating or simulating music with a computer.

The speed of sound can vary considerably, depending on the material, the temperature, the humidity, and so on. For example, in dry air at room temperature (68 degrees Farenheit), sound travels at a rate of 1,125 feet (343 meters) per second, or 768 miles (1,236 kilometers) per hour. Perhaps surprisingly, the speed of sound varies little with respect to air pressure, although it does vary with temperature.

The reflection and absorbtion of sound is a much more difficult topic, since it depends so much on the material, the shape and thickness of the material, and the frequency of the sound. Modeling well the acoustics of a concert hall, for example, is quite challenging. To understand how much such reflections can affect the overall sound that we hear, consider a concert hall that is 200 feet long and 100 feet wide. Based on the speed of sound given above, it will take a sound wave ${}^{2\times 200}/{}_{1125} = 0.355$ seconds to travel from the front of the room to the back of the room and back to the front again. That ${}^{1}/{}_{3}$ of a second, if loud enough, would result in a significant distortion of the music, and corresponds to about one beat with a metronome set at 168.

With respect to our interpretation of music, sound has (at least) three key properties:

1. *Frequency* (perceived as *pitch*).
2. *Amplitude* (perceived as *loudness*).
3. *Spectrum* (perceived as *timbre*).

We discuss each of these in the sections that follow.

18.1.1 Frequency and Period

The *frequency* f is simply the rate of the vibrations (or repetitions, or cycles) of the sound, and is the inverse of the *period* (or duration, or wavelength) p of each of the vibrations:

$$f = \frac{1}{p}$$

Frequency is measured in *Hertz* (abbreviated Hz), where 1 Hz is defined as one cycle per second. For example, the sound wave in Figure 18.1 has a frequency of 1000 Hz (i.e. 1 kHz) and a period of $1/1000$ second (i.e. 1 ms).

In trigonometry, functions like sine and cosine are typically applied to angles that range from 0 to 360 degrees. In audio processing (and signal processing in general) angles are instead usually measured in *radians*, where 2π radians is equal to $360°$. Since the sine function has a period of 2π and a frequency of $1/2\pi$, it repeats itself every 2π radians:

$$\sin(2\pi k + \theta) = \sin\theta$$

for any integer k.

But for our purposes it is better to parameterize these functions over frequency as follows. Since $\sin(2\pi t)$ covers one full cycle in one second, i.e. has a frequency of 1 Hz, it makes sense that $\sin(2\pi f t)$ covers f cycles in one second, i.e. has a frequency of f. Indeed, in signal processing the quantity ω is defined as:

$$\omega = 2\pi f$$

That is, a pure sine wave as a function of time behaves as $\sin(\omega t)$.

Finally, it is convenient to add a *phase* (or *phase angle*) to our formula, which effectively shifts the sine wave in time. The phase is usually represented by ϕ. Adding a multiplicative factor A for amplitude (see next section), we arrive at our final formula for a sine wave as a function of time:

$$s(t) = A\sin(\omega t + \phi)$$

A negative value for ϕ has the effect of "delaying" the sine wave, whereas a positive value has the effect of "starting early." Note also that this equation holds for negative values of t.

All of the above can be related to cosine by recalling the following identity:

$$\sin(\omega t + \frac{\pi}{2}) = \cos(\omega t)$$

More generally:

$$A\sin(\omega t + \phi) = a\cos(\omega t) + b\sin(\omega t)$$

Given a and b we can solve for A and ϕ:

$$\begin{aligned} A &= \sqrt{a^2 + b^2} \\ \phi &= \tan^{-1}\frac{b}{a} \end{aligned}$$

Given A and ϕ we can also solve for a and b:

$$\begin{aligned} a &= A\cos(\phi) \\ b &= A\sin(\phi) \end{aligned}$$

18.1.2 Amplitude and Loudness

Amplitude can be measured in several ways. The *peak amplitude* of a signal is its maximum deviation from zero; for example our sine wave in Figure 18.1 has a peak amplitude of 1. But different signals having the same peak amplitude have more or less "energy," depending on their "shape." For example, Figure 18.2 shows four kinds of signals: a sine wave, a square wave, a sawtooth wave, and a triangular wave (whose names are suitably descriptive). Each of them has a peak amplitude of 1. But, intuitively, one would expect the square wave, for example, to have more "energy," or "power," than a sine wave, because it is "fatter." In fact, it's value is everywhere either +1 or -1.

To measure this characteristic of a signal, scientists and engineers often refer to the *root-mean-square* amplitude, or RMS. Mathematically, the root-mean-square is the square root of the mean of the squared values of a given quantity. If x is a discrete quantity given by the values $x_1, x_2, ..., x_n$, the formula for RMS is:

$$x_{\mathrm{RMS}} = \sqrt{\frac{x_1^2 + x_2^2 + ... + x_n^2}{n}}$$

And if f is continuous function, its RMS value over the interval $T_1 \leqslant t \leqslant T_2$ is given by:

$$\sqrt{\frac{1}{T_2 - T_1}\int_{-T_1}^{T_2} f(t)^2 dt}$$

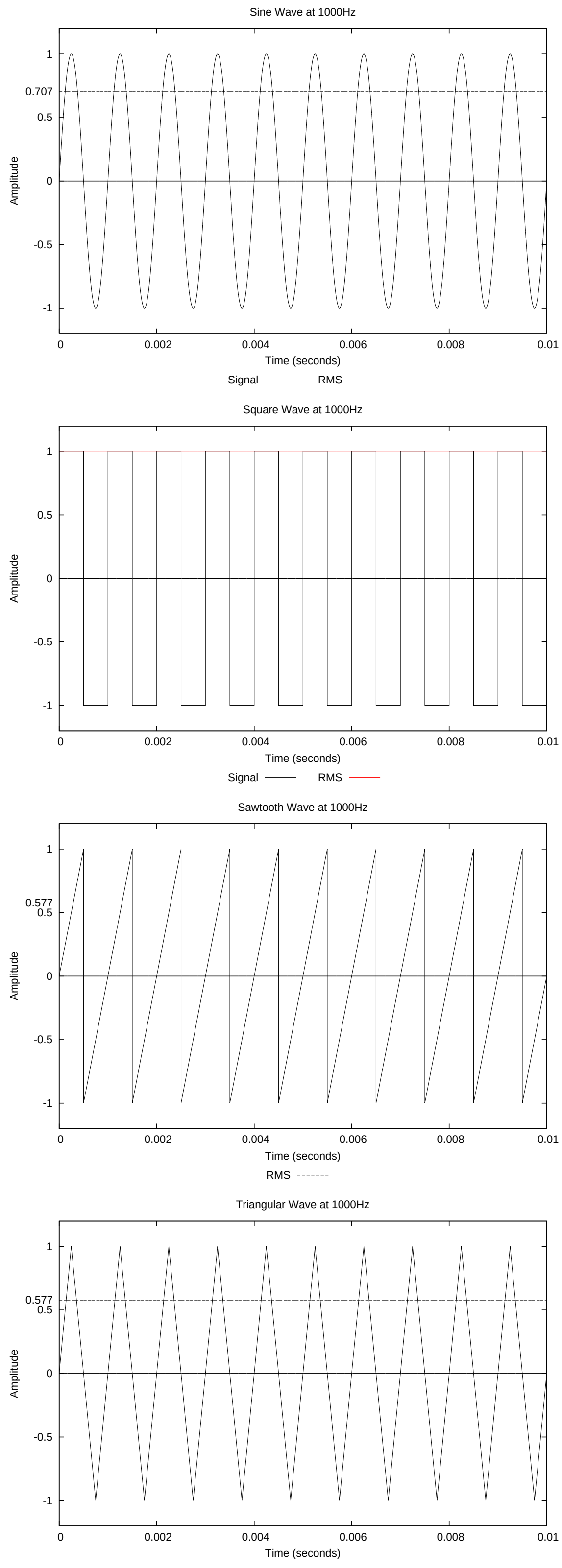


Figure 18.2: RMS Amplitude for Different Signals

For a sine wave, it can be shown that the RMS value is approximately 0.707 of the peak value. For a square wave, it is 1.0. And for both a sawtooth wave and a triangular wave, it is approximately 0.577. Figure 18.2 shows these RMS values superimposed on each of the four signals.

Another way to measure amplitude is to use a relative logarithmic scale that more aptly reflects how we hear sound. This is usually done by measuring the sound level (usually in RMS) with respect to some reference level. The number of *decibels* (dB) of sound is given by:

$$S_{dB} = 10 \log_{10} \frac{S}{R}$$

where S is the RMS sound level, and R is the RMS reference level. The accepted reference level for the human ear is 10^{-12} watts per square meter, which is roughly the threshold of hearing.

A related concept is the measure of how much useful information is in a signal relative to the "noise." The *signal-to-noise ratio*, or *SNR*, is defined as the ratio of the *power* of each of these signals, which is the square of the RMS value:

$$SNR = \left(\frac{S}{N}\right)^2$$

where S and N are the RMS values of the signal and noise, respectively. As is often the case, it is better to express this on a logarithmic scale, as follows:

$$\begin{aligned} SNR_{dB} &= 10 \log_{10} \left(\frac{S}{N}\right)^2 \\ &= 20 \log_{10} \frac{S}{N} \end{aligned}$$

The *dynamic range* of a system is the difference between the smallest and largest values that it can process. Because this range is often very large, it is usually measured in decibels, which is a logarithmic quantity. The ear, for example, has a truly remarkable dynamic range—about 130 dB. To get some feel for this, silence should be considered 0 dB, a whisper 30 dB, normal conversation about 60 dB, loud music 80 dB, a subway train 90 dB, and a jet plane taking off or a very loud rock concert 120 dB or higher.

Note that if you double the sound level, the decibels increase by about 3 dB, whereas a million-fold increase corresponds to 60 dB:

$$\begin{array}{lclcl} 10 \log_{10} 2 & = & 10 \times 0.301029996 & \cong & 3 \\ 10 \log_{10} 10^6 & = & 10 \times 6 & = & 60 \end{array}$$

So the ear is truly adaptive! (The eye also has a large dynamic range with respect to light intensity, but not quite as much as the ear, and its response time is much slower.)

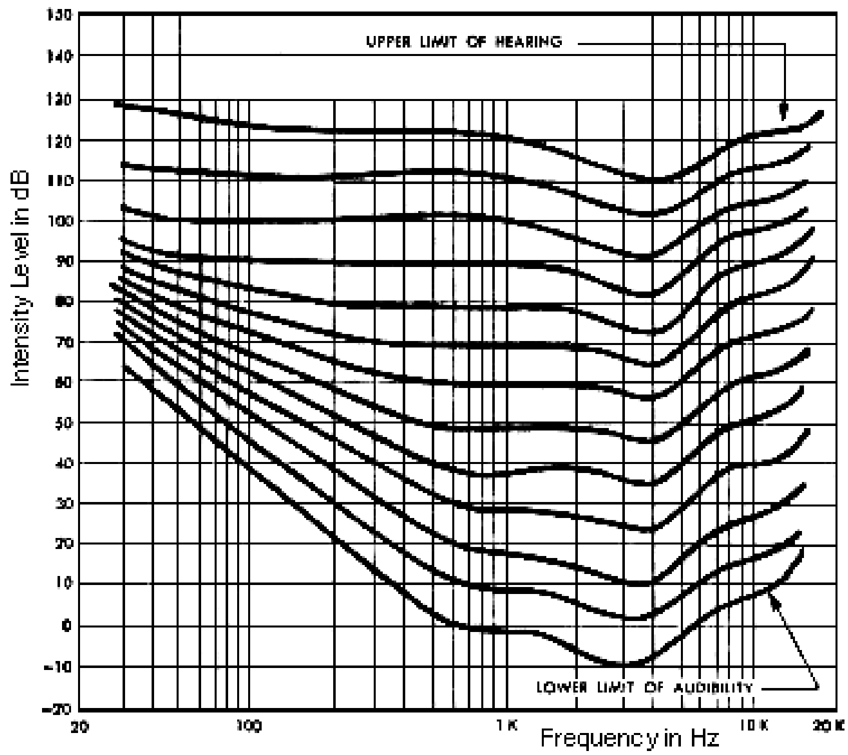


Figure 18.3: Fletcher-Munson Equal Loudness Contour

Loudness is the perceived measure of amplitude, or volume, of sound, and is thus subjective. It is most closely aligned with RMS amplitude, with one important exception: loudness depends somewhat on frequency! Of course that's obvious for really high and really low frequencies (since at some point we can't hear them at all), but in between things aren't constant either. Furthermore, no two humans are the same. Figure 18.3 shows the *Fletcher-Munson Equal-Loudness Contour*, which reflects the perceived equality of sound intensity by the average human ear with respect to frequency. Note from this figure that:

- The human ear is less sensitive to low frequencies.
- The maximum sensitivity is around 3-4 kHz, which roughly corre-

sponds to the resonance of the auditory canal.

Another important psychoacoustical property is captured in the *Weber-Fechner Law*, which states that the *just noticeable difference* (jnd) in a quantity—i.e. the minimal change necessary for humans to notice something in a cognitive sense—is a relative constant, independent of the absolute level. That is, the ratio of the change to the absolute measure of that quantity is constant:

$$\frac{\Delta q}{q} = k$$

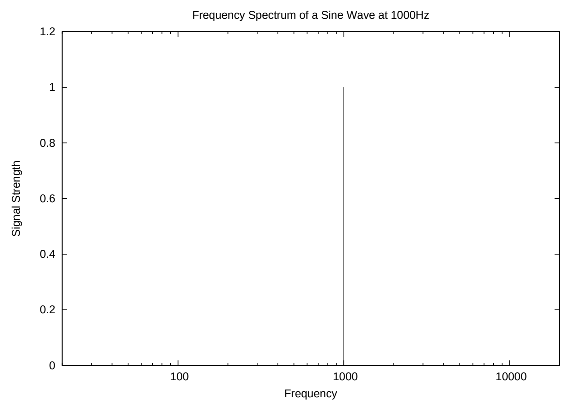
The jnd for loudness happens to be about 1 db, which is another reason why the decibel scale is so convenient. 1 db corresponds to a sound level ratio of 1.25892541. So, in order for a person to "just notice" an increase in loudness, one has to increase the sound level by about 25%. If that seems high to you, it's because your ear is so adaptive that you are not even aware of it.

18.1.3 Frequency Spectrum

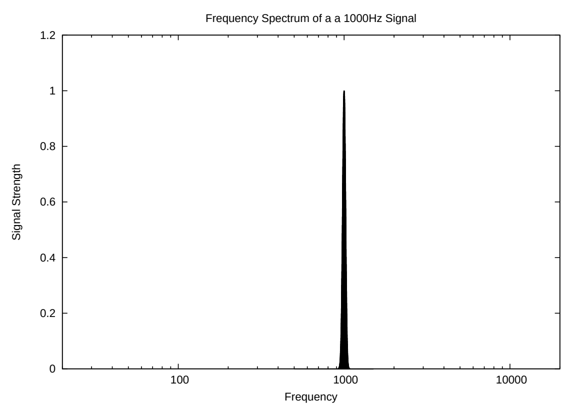
Humans can hear sound approximately in the range 20 Hz to 20,000 Hz = 20 kHz. This is a dynamic range in frequency of a factor of 1000, or 30 dB. Different people can hear different degrees of this range (I can hear very low tones well, but not very high ones). On a piano, the fundamental frequency of the lowest note is 27.5 Hz, middle (concert) A is 440 hz, and the topmost note is about 4 kHz. Later we will learn that these notes also contain *overtones*—multiples of the fundamental frequency—that contribute to the *timbre*, or sound quality, that distinguishes one instrument from another. (Overtones are also called *harmonics* or *partials*.)

The *phase*, or time delay, of a signal is important too, and comes into play when we start mixing signals together, which can happen naturally, deliberately, from reverberations (room acoustics), and so on. Recall that a pure sine wave can be expressed as $\sin(\omega t + \phi)$, where ϕ is the *phase angle*. Manipulating the phase angle is common in additive synthesis and amplitude modulation, topics to be covered in later chapters.

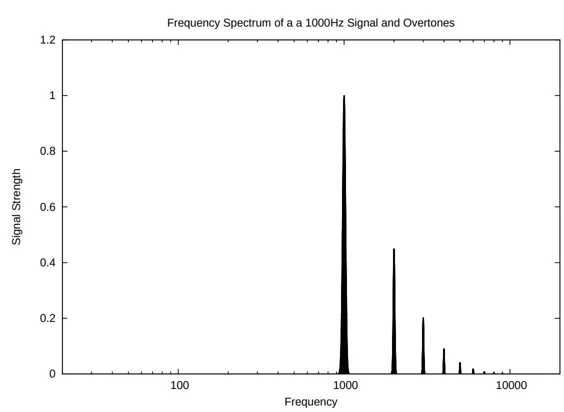
A key point is that most sounds do not consist of a single, pure sine wave—rather, they are a combination of many frequencies, and at varying



(a) Spectral plot of pure sine wave



(b) Spectral plot of a noisy sine wave



(c) Spectral plot of a musical tone

Figure 18.4: Spectral Plots of Different Signals

phases relative to one another. Thus it is helpful to talk of a signal's *frequency spectrum*, or spectral content. If we have a regular repetitive sound (called a *periodic signal*) we can plot its spectral content instead of its time-varying graph. For a pure sine wave, this looks like an impulse function, as shown in Figure 18.4a.

But for a richer sound, it gets more complicated. First, the distribution of the energy is not typically a pure impulse, meaning that the signal might vary slightly above and below a particular frequency, and thus its frequency spectrum typically looks more like Figure 18.4b.

In addition, a typical sound has many different frequencies associated with it, not just one. Even for an instrument playing a single note, this will include not just the perceived pitch, which is called the *fundamental frequency*, but also many *overtones* (or harmonics) which are multiples of the fundamental, as shown in Figure 18.4c. The *natural harmonic series* is one that is approximated often in nature, and has a harmonically decaying series of overtones.

What's more, the articulation of a note by a performer on an instrument causes these overtones to vary in relative size over time. There are several ways to visualize this graphically, and Figure 18.5 shows two of them. In 18.5a, shading is used to show the varying amplitude over time. And in 18.5b, a 3D projection is used.

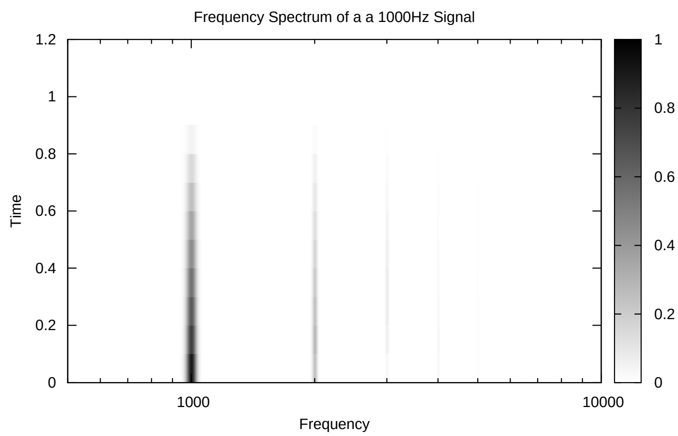
The precise blend of the overtones, their phases, and how they vary over time, is primarily what distinguishes a particular note, say concert A, on a piano from the same note on a guitar, a violin, a saxophone, and so on. We will have much more to say about these issues in later chapters.

[See pictures at:
`http://www.computermusicresource.com/spectrum.html`.]

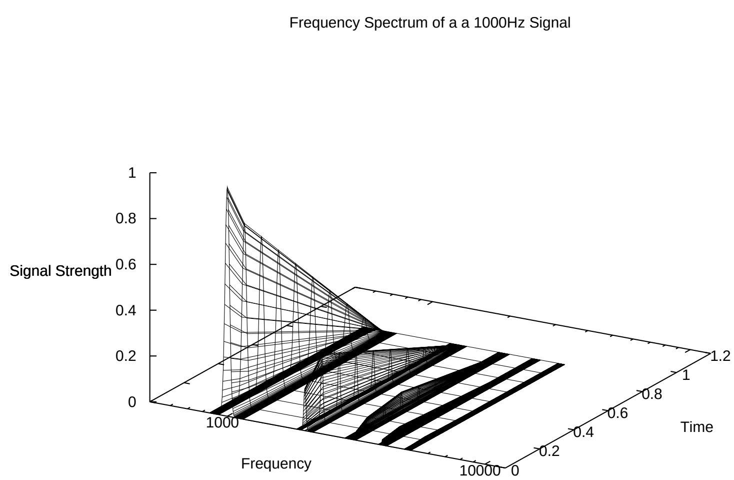
18.2 Digital Audio

The preceding discussion has assumed that sound is a continuous quantity, which of course it is, and thus we represent it using continuous mathematical functions. If we were using an analog computer, we could continue with this representation, and create electronic music accordingly. Indeed, the earliest electronic synthesizers, such as the *Moog synthesizer* of the 1960's, were completely analog.

However, most computers today are *digital*, which require representing



(a) Using shading



(b) Using 3D projection

Figure 18.5: Time-Varying Spectral Plots

sound (or signals in general) using digital values. The simplest way to do this is to represent a continuous signal as a *sequence of discrete samples* of the signal of interest. An *analog-to-digital converter*, or ADC, is a device that converts an instantaneous sample of a continuous signal into a binary value. The microphone input on a computer, for example, connects to an ADC.

Normally the discrete samples are taken at a fixed *sampling rate*. Choosing a proper sampling rate is quite important. If it is too low, we will not acquire sufficient samples to adequately represent the signal of interest. And if the rate is too high, it may be an overkill, thus wasting precious computing resources (in both time and memory consumption). Intuitively, it seems that the highest frequency signal that we could represent using a sampling rate r would have a frequency of $r/2$, in which case the result would have the appearance of a square wave, as shown in Figure 18.6a. Indeed, it is easy to see that problems could arise if we sampled at a rate significantly lower than the frequency of the signal, as shown in Figures 18.6b and 18.6c for sampling rates equal to, and one-half, of the frequency of the signal of interest—in both cases the result is a sampled signal of 0 Hz!

Indeed, this observation is captured in what is known as the *Nyquist-Shannon Sampling Theorm* that, stated informally, says that the accurate reproduction of an analog signal (no matter how complicated) requires a sampling rate that is at least twice the highest frequency of the signal of interest.

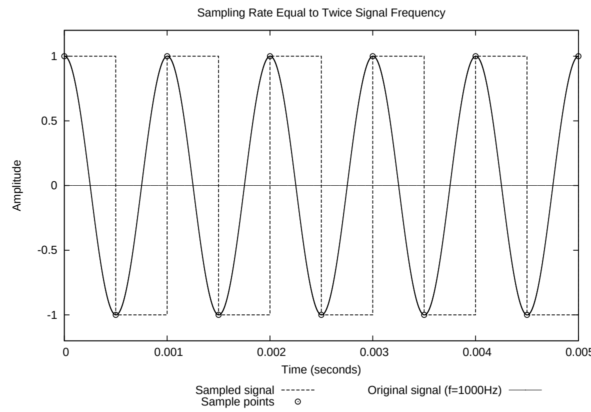
For example, for audio signals, if the highest frequency humans can hear is 20 kHz, then we need to sample at a rate of at least 40 kHz for a faithful reproduction of sound. In fact, CD's are recorded at 44.1 kHz. But many people feel that this rate is too low, as some people can hear beyond 20 kHz. Another recording studio standard is 48 kHz. Interestingly, a good analog tape recorder from generations ago was able to record signals with frequency content even higher than this—perhaps digital is not always better!

18.2.1 From Continuous to Discrete

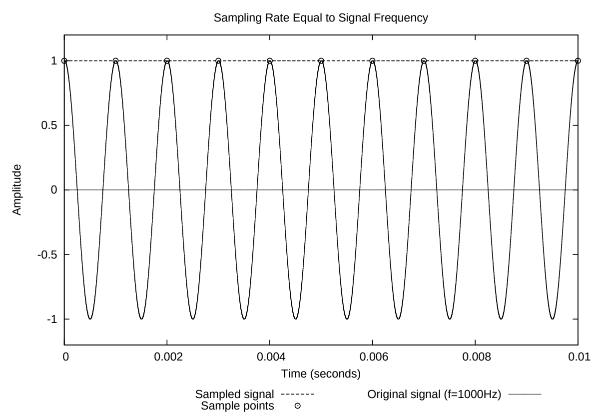
Recall the definition of a sine wave from Section 18.1.1:

$$s(t) = A\sin(\omega t + \phi)$$

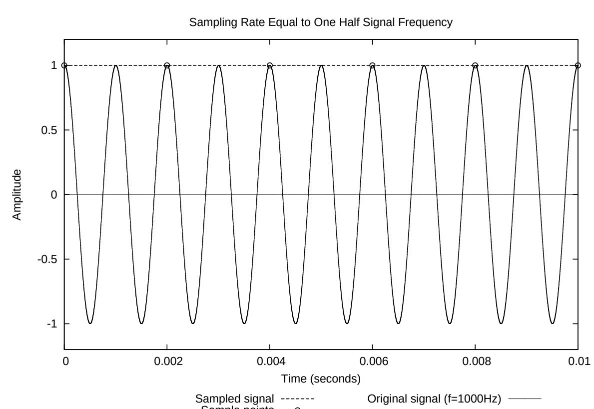
We can easily and intuitively convert this to the discrete domain by replacing the time t with the quantity n/r, where n is the integer index into the



(a)



(b)



(c)

Figure 18.6: Choice of Sampling Rate

sequence of discrete samples, and r is the sampling rate discussed above. If we use $s[n]$ to denote the $(n+1)^{\text{th}}$ sample of the signal, we have:

$$s[n] = A\sin\left(\frac{\omega n}{r} + \phi\right), \qquad n = 0, 1, ..., \infty$$

Thus $s[n]$ corresponds to the signal's value at time $^n/_r$.

18.2.2 Fixed-Waveform Table-Lookup Synthesis

One of the most fundamental questions in digital audio is how to generate a sine wave as efficiently as possible, or, in general, how to generate a fixed periodic signal of any form (sine wave, square wave, sawtooth wave, even a sampled sound bite). A common and efficient way to generate a periodic signal is through *fixed-waveform table-lookup synthesis*. The idea is very simple: store in a table the samples of a desired periodic signal, and then index through the table at a suitable rate to reproduce that signal at some desired frequency. The table is often called a *wavetable*.

In general, if we let:

$$\begin{aligned} L &= \text{table length} \\ f &= \text{resulting frequency} \\ i &= \text{indexing increment} \\ r &= \text{sample rate} \end{aligned}$$

then we have:

$$f = \frac{ir}{L}$$

For example, suppose the table contains 8196 samples. If the sample rate is 44.1 kHz, how do we generate a tone of, say, 440 Hz? Plugging in the numbers and solving the above equation for i, we get:

$$\begin{aligned} 440 &= \frac{i \times 44.1\text{kHz}}{8196} \\ i &= \frac{440 \times 8196}{44.1\text{kHz}} \\ &= 81.77 \end{aligned}$$

So, if we were to sample approximately every 81.77^{th} value in the table, we would generate a signal of 440 Hz.

Now suppose the table T is a vector, and $T[n]$ is the nth element. Let's call the exact index increment i into a continuous signal the *phase*, and the actual index into the corresponding table the *phase index* p. The computation of successive values of the phase index and output signal s is then captured by these equations:

$$
\begin{aligned}
p_o &= \lfloor \phi_0 + 0.5 \rfloor \\
p_{n+1} &= (p_n + i) \bmod L \\
s_n &= T[\ \lfloor p_n + 0.5 \rfloor\ ]
\end{aligned}
$$

$\lfloor a+0.5 \rfloor$ denotes the floor of $a+0.5$, which effectively rounds a to the nearest integer. ϕ_0 is the initial phase angle (recall earlier discussion), so p_0 is the initial index into the table that specifies where the fixed waveform should begin.

Instead of rounding the index, one could do better by *interpolating* between values in the table, at the expense of efficiency. In practice, rounding the index is often good enough. Another way to increase accuracy is to simply increase the size of the table.

18.2.3 Aliasing

Earlier we saw examples of problems that can arise if the sampling rate is not high enough. We saw that if we sample a sine wave at twice its frequency, we can suitably capture that frequency. If we sample at exactly its frequency, we get 0 Hz. But what happens in between? Consider a sampling rate ever-so-slightly higher or lower than the sine wave's fundamental frequency--in both cases, this will result in a frequency much lower than the original signal, as shown in Figures 18.7 and 18.8. This is analogous to the effect of seeing spinning objects under fluorescent or LED light, or old motion pictures of the spokes in the wheels of horse-drawn carriages.

These figures suggest the following. Suppose that m is one-half the sampling rate. Then:

Original signal	Reproduced signal
$0 - m$	$0 - m$
$m - 2m$	$m - 0$
$2m - 3m$	$0 - m$
$3m - 4m$	$m - 0$
$\cdots$	$\cdots$

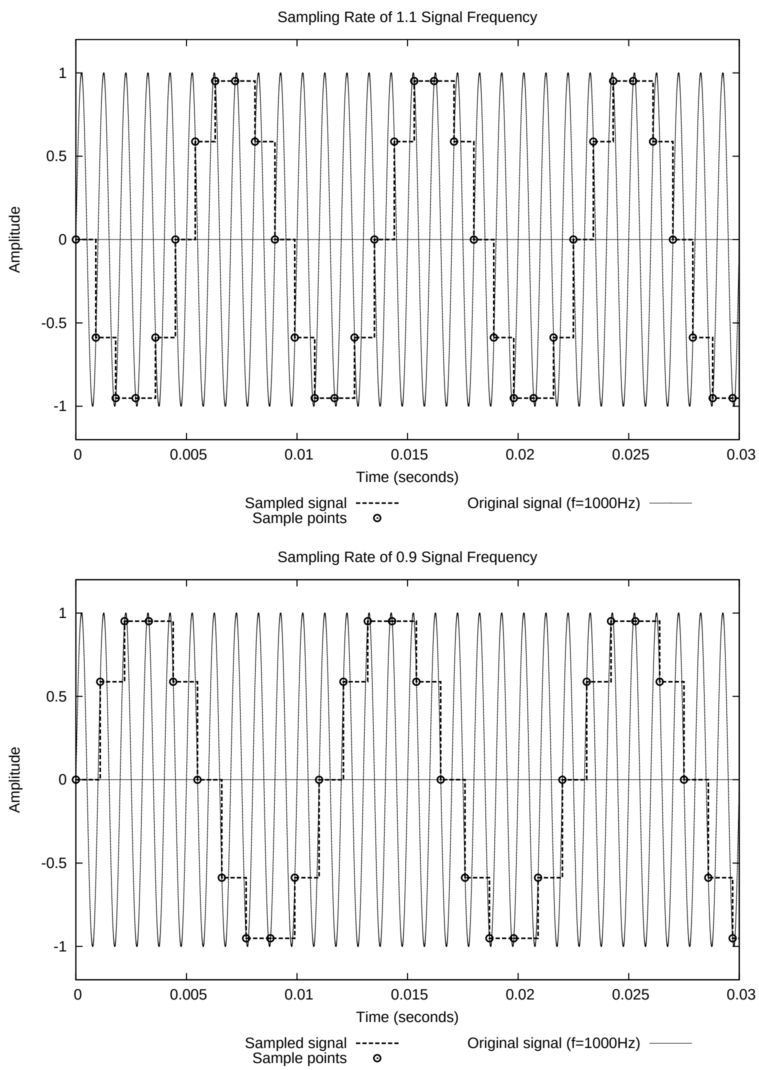


Figure 18.7: Aliasing 1

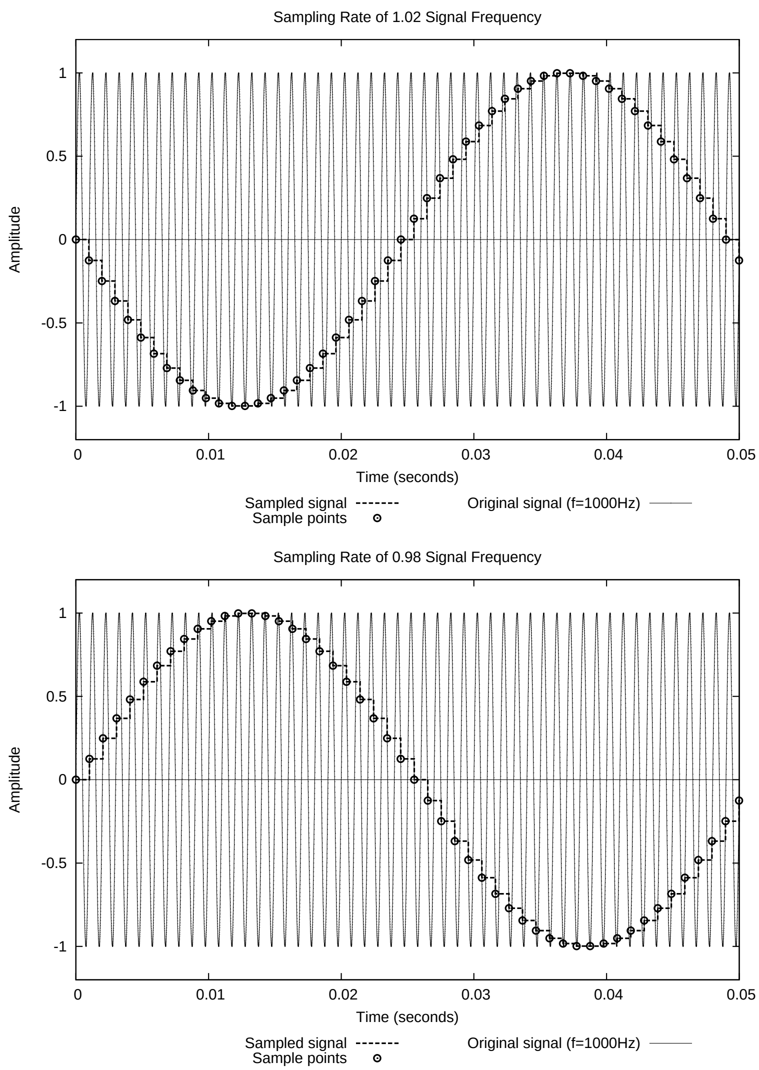


Figure 18.8: Aliasing 2

This phenomenon is called *aliasing*, or *foldover* of the signal onto itself.

This is not good! In particular, it means that audio signals in the ultrasonic range will get "folded" into the audible range. To solve this problem, we can add an analog *low-pass filter* in front of the ADC–usually called an *anti-aliasing* filter—to eliminate all but the audible sound before it is digitized. In practice, however, this can be tricky. For example, a steep analog filter introduces *phase distortion* (i.e. frequency-dependent time delays), and early digital recordings were notorious in the "harsh sound" that resulted. This can be fixed by using a filter with less steepness (but resulting in more aliasing), or using a time correlation filter to compensate, or using a technique called *oversampling*, which is beyond the scope of this text.

A similar problem occurs at the other end of the digital audio process—i.e. when we reconstruct an analog signal from a digital signal using a *digital-to-analog converter*, or DAC. The digital representation of a signal can be viewed mathematically as a stepwise approximation to the real signal, as shown in Figure 18.9, where the sampling rate is ten times the frequency of interest. As discussed earlier, at the highest frequency (i.e. at one-half the sampling rate), we get a square wave. As we will see in Chapter 20, a square wave can be represented mathematically as the sum of an infinite sequence of sine waves, consisting of the fundamental frequency and all of its odd harmonics. These harmonics can enter the ultrasonic region, causing potential havoc in the analog circuitry, or in a dog's ear (dogs can hear frequencies much higher than humans). The solution is to add yet another low-pass filter, called an *anti-imaging* or *smoothing* filter to the output of the DAC. In effect, this filter "connects the dots," or interpolates, between successive values of the stepwise approximation.

In any case, a basic block diagram of a typical digital audio system—from sound input to sound output—is shown in Figure 18.10.

18.2.4 Quantization Error

In terms of amplitude, remember that we are using digital numbers to represent an analog signal. For conventional CD's, 16 bits of precision are used. If we were to compute and then "listen to" the round-off errors that are induced, we would hear subtle imperfections, called *quantization error*, or more commonly, "noise."

One might compare this to "hiss" on a tape recorder (which is due to the molecular disarray of the magnetic recording medium), but there are

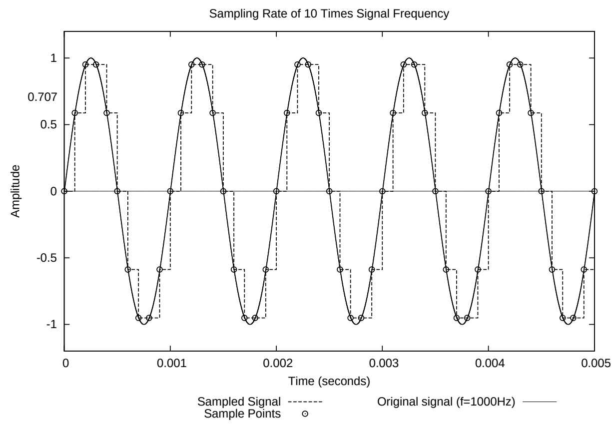


Figure 18.9: A Properly Sampled Signal

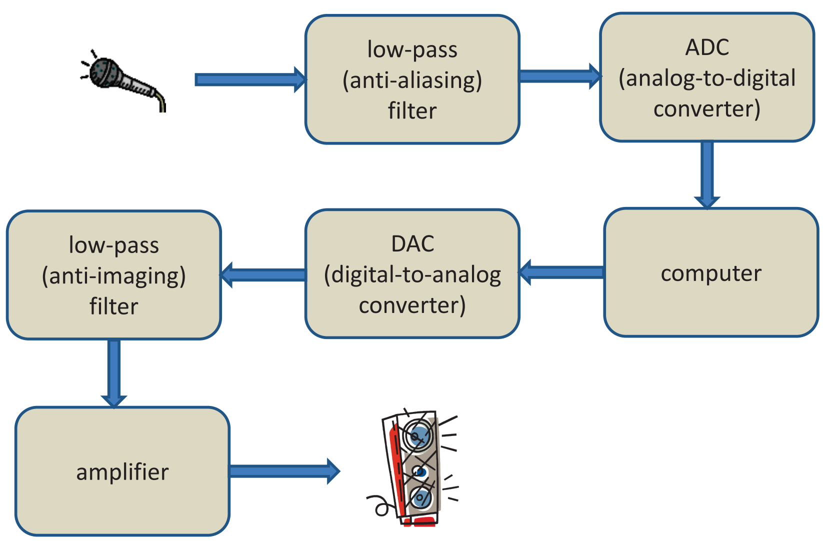


Figure 18.10: Block Diagram of Typical Digital Audio System

important differences. First of all, when there is no sound, there is no quantization error in a digital signal, but there is still hiss on a tape. Also, when the signal is very low and regular, the quantization error becomes somewhat regular as well, and is thus audible as something different from hiss. Indeed, it's only when the signal is loud and complex that quantization error compares favorably to tape hiss.

One solution to the problem of low signal levels mentioned above is to purposely introduce noise into the system to make the signal less predictable. This fortuitous use of noise deserves a better name, and indeed it is called *dither*.

18.2.5 Dynamic Range

What is the dynamic range of an n-bit digital audio system? If we think of quantization error as noise, it makes sense to use the equation for SNR_{dB} given in Section 18.1.2:

$$SNR_{dB} = 20\log_{10}\frac{S}{N}$$

But what should N be, i.e. the quantization error? Given a signal amplitude range of $\pm a$, with n bits of resolution it is divided into $2a/2^n$ points. Therefore the dynamic range is:

$$\begin{aligned} 20\log_{10}\left(\frac{2a}{2a/2^n}\right) &= 20\times\log_{10}(2^n) \\ &= 20\times n\times\log_{10}(2) \\ &\approx 20\times n\times(0.3) \\ &= 6n \end{aligned}$$

For example, a 16-bit digital audio system results in a dynamic range of 96 dB, which is pretty good, although a 20-bit system yields 120 dB, corresponding to the dynamic range of the human ear.

Exercise 18.1 For each of the following, say whether it is a longitudinal wave or a transverse wave:

- A vibrating violin string.
- Stop-and-go traffic on a highway.

- "The wave" in a crowd at a stadium.
- "Water hammer" in the plumbing of your house.
- The wave caused by a stone falling in a pond.
- A radio wave.

Exercise 18.2 You see a lightning strike, and 5 seconds later you hear the thunder. How far away is the lightning?

Exercise 18.3 You clap your hands in a canyon, and 2 seconds later you hear an echo. How far away is the canyon wall?

Exercise 18.4 By what factor must one increase the RMS level of a signal to yield a 10 dB increase in sound level?

Exercise 18.5 A dog can hear in the range 60-45,000 Hz, and a bat 2,000-110,000 Hz. In terms of the frequency response, what are the corresponding dynamic ranges for these two animals, and how do they compare to that of humans?

Exercise 18.6 What is the maximum number of audible overtones in a note whose fundamental frequency is 100 Hz? 500 Hz? 1500 Hz? 5 kHz?

Exercise 18.7 Consider a continuous input signal whose frequency is f. Devise a formula for the frequency r of the reproduced signal given a sample rate s.

Exercise 18.8 How much memory is needed to record 3 minutes of stereo sound using 16-bit samples taken at a rate of 44.1 kHz?

Exercise 18.9 If we want the best possible sound, how large should the table be using fixed-waveform table-lookup synthesis, in order to cover the audible frequency range?

Exercise 18.10 The Doppler effect occurs when a sound source is in motion. For example, as a police car moves toward you its siren sounds higher than it really is, and as it goes past you, it gets lower. How fast would a police car have to go to change a siren whose frequency is the same as concert A, to a pitch an octave higher? (i.e. twice the frequency) At that speed, what frequency would we hear after the police car passes us?

Chapter 19

Euterpea's Signal Functions

```
{-# LANGUAGE Arrows #-}
module Euterpea.Examples.SigFuns where
import Euterpea
import Control.Arrow ((>>>), (<<<), arr)
```

Details: The first line in the module header above is a *compiler pragma*, amd in this case is telling GHC to accept *arrow syntax*, which will be explained in Section 19.1.

In this chapter we show how the theoretical concepts involving sound and signals studied in the last chapter are manifested in Euterpea. The techniques learned will lay the groundwork for doing two broad kinds of activities: *sound synthesis* and *audio processing*. Sound synthesis might include creating the sound of a footstep on dry leaves, simulating a conventional musical instrument, creating an entirely new instrument sound, or composing a single "soundscape" that stands alone as a musical composition. Audio processing includes such things as equalization, filtering, reverb, special effects, and so on. In future chapters we will study various techniques for achieving these goals.

19.1 Signals and Signal Functions

As we saw in Chapter 17, it would seem natural to represent a signal as an abstract type, say *Signal T* in Haskell, and then define functions to add, multiply, take the sine of, and so on, signals represented in this way. For example, *Signal Float* would be the type of a time-varying floating-point number, *Signal AbsPitch* would be the type of a time-varing absolute pitch, and so on. Then given $s_1, s_2 :: \mathit{Signal\ Float}$ we might simply write $s_1 + s_2$, $s_1 * s_2$, and $\mathit{sin}\ s_1$ as examples of applying the above operations. Haskell's numeric type class hierarchy would make this particularly easy to do. Indeed, several domain-specific languages based on this approach have been defined before, beginning with the language *Fran* [EH97] that was designed for writing computer animation programs.

But years of experience and theoretical study have revealed that such an approach leads to a language with subtle time- and space-leaks,[1] for reasons that are beyond the scope of this textbook [LH07]. Therefore Euterpea takes a somewhat different approach, as described below.

Perhaps the simplest way to understand Euterpea's approach to programming with signals is to think of it as a language for expressing *signal processing diagrams* (or equivalently, electrical circuits). We refer to the lines in a typical signal processing diagram as *signals*, and the boxes that convert one signal into another as *signal functions*. For example, this very simple diagram has two signals, x and y, and one signal function, *sigfun*:

Image not in repository!

Using Haskell's *arrow syntax* [Hug00, Pat01], this diagram can be expressed as a code fragment in Euterpea simply as:

$$y \leftarrow \mathit{sigfun} \prec x$$

Details: The syntax $\leftarrow$ and $\prec$ is typeset here in an attractive way, but the user will have to type `<-` and `-<`, respectively, in her source file.

[1] A time-leak in a real-time system occurs whenever a time-dependent computation falls behind the current time because its value or effect is not needed yet, but then requires "catching up" at a later point in time. This catching up process can take an arbitrarily long time, and may consume additional space as well. It can destroy any hope for real-time behavior if not managed properly.

Arrows and arrow syntax will be described in much more detail in Chapter **??**. For now, keep in mind that $\leftarrow$ and $\prec$ are part of the *syntax*, and are not simply binary operators. Indeed, we can't just write the above code fragment anywhere. It has to be within an enclosing **proc** construct whose result type is that of a signal function. The **proc** construct begins with the keyword **proc** along with an argument, analogous to an anonymous function. For example, a signal function that takes a signal of type *Double* and adds 1 to every signal sample, and then applies *sigfun* to the resulting signal, can be written:

$$
\begin{array}{l}
\textbf{proc}\ y \rightarrow \textbf{do} \\
\quad x \leftarrow \mathit{sigfun} \prec y + 1 \\
\quad \mathit{outA} \prec x
\end{array}
$$

> **Details:** The **do** keyword in arrow syntax introduces layout, just as it does in monad syntax.

Note the analogy of this code to the following snippet involving an ordinary anonymous function:

$$
\begin{array}{l}
\lambda y \rightarrow \\
\quad \textbf{let}\ x = \mathit{sigfun}\ (y + 1) \\
\quad \textbf{in}\ x
\end{array}
$$

The important difference, however, is that *sigfun* works on a signal, which we can think of as a stream of values, whose representative values at the "point" level are the variables x and y above. So in reality we would have to write something like this:

$$
\begin{array}{l}
\lambda \mathit{ys} \rightarrow \\
\quad \textbf{let}\ \mathit{xs} = \mathit{sigfun}\ (\mathit{map}\ (+1)\ \mathit{ys}) \\
\quad \textbf{in}\ \mathit{xs}
\end{array}
$$

to achieve the effect of the arrow code above. The arrow syntax allows us to avoid worrying about the streams themselves. It also has other important advantages that are beyond the scope of the current discussion.

Arrow syntax is just that--syntactic sugar that is expanded into a set of conventional functions that work just as well, but are more cumbersome to program with (just as with monad syntax). This syntactic expansion will be described in more detail in Chapter **??**. To use the arrow syntax within a ".lhs" file, one must declare a compiler flag in GHC at the very beginning of the file, as follows:

{-# LANGUAGE Arrows #-}

19.1.1 The Type of a Signal Function

Polymorphically speaking, a signal function has type:

$Clock\ c \Rightarrow SigFun\ c\ a\ b$

which should be read, "for some clock type (i.e. sampling rate) c, this is the type of signal functions that convert signals of type a into signals of type b."

The type variable c indicates what clock rate is being used, and for our purposes will always be one of two types: *AudRate* or *CtrRate* (for *audio rate* and *control rate*, respectively). Being able to express the sampling rate of a signal function is what we call *clock polymorphism.* Although we like to think of signals as continuous, time-varying quantities, in practice we know that they are sampled representations of continous quantities, as discussed in the last chapter. However, some signals need to be sampled at a very high rate—say, an audio signal—whereas other signals need not be sampled at such a high rate—say, a signal representing the setting of a slider. The problem is, we often want to mix signals sampled at different rates; for example, the slider might control the volume of the audio signal.

One solution to this problem would be to simply sample everything at the very highest rate, but this is computationally inefficient. A better approach is to sample signals at their most appropriate rate, and to perform coercions to "up sample" or "down sample" a signal when it needs to be combined with a signal sampled at a different rate. This is the approach used in Euterpea.

More specifically, the base type of each signal into and out of a signal function must satisfy the type class constraint *Clock* c, where c is a *clock type*. The *Clocked* class is defined as:

```
class Clock c where
  rate :: c → Double
```

The single method *rate* allows the user to extract the sampling rate from the type. In Euterpea, the *AudRate* is pre-defined to be 44.1 kHz, and the *CtrRate* is set at 4.41 kHz. Here are the definitions of *AudRate* and *CtrRate*, along with their instance declarations in the *Clock* class, to achieve this:

```
data AudRate
data CtrRate
```

$$\textbf{instance}\ \mathit{Clock}\ \mathit{AudRate}\ \textbf{where}$$
$$\quad \mathit{rate}\ \_ = 44100$$
$$\textbf{instance}\ \mathit{Clock}\ \mathit{CtrRate}\ \textbf{where}$$
$$\quad \mathit{rate}\ \_ = 4410$$

Because these two clock types are so often used, it is helpful to define a couple of type synonyms:

$$\textbf{type}\ \mathit{AudSF}\ a\ b = \mathit{SigFun}\ \mathit{AudRate}\ a\ b$$
$$\textbf{type}\ \mathit{CtrSF}\ a\ b\ = \mathit{SigFun}\ \mathit{CtrRate}\ a\ b$$

From these definitions it should be clear how to define your own clock type.

Details: Note that $AudRate$ and $CtrRate$ have no constructors—they are called *empty* data types. More precisely, they are each inhabited by exactly one value, namely $\perp$.

The sampling rate can be determined from a given clock type. In this way, a coercion function can be written to change a signal sampled at one rate to a signal sampled at some other rate. In Euterpea, there are two such functions that are pre-defined:

$$\mathit{coerce}, \mathit{upsample} :: (\mathit{Clock}\ c_1, \mathit{Clock}\ c_2) \Rightarrow$$
$$\mathit{SigFun}\ c_1\ a\ b \rightarrow \mathit{SigFun}\ c_2\ a\ b$$

The function *coerce* looks up the sampling rates of the input and output signals from the type variables c_1 and c_2. It then either stretches the input stream by duplicating the same element or contracts it by skipping elements. (It is also possible to define a more accurate coercion function that performs interpolation, at the expense of performance.)

For simpler programs, the overhead of calling *coerce* might not be worth the time saved by generating signals with lower resolution. (Haskells fractional number implementation is relatively slow.) The specialized coercion function *upsample* avoids this overhead, but only works properly when the output rate is an integral multiple of the input rate (which is true in the case of *AudRate* and *CtrRate*).

Keep in mind that one does not have to commit a signal function to a particular clock rate—it can be left *polymorphic*. Then that signal function will adapt its sampling rate to whatever is needed in the context in which it is used.

Also keep in mind that a signal function is an abstract function. You cannot just apply it to an argument like an ordinary function—that is the purpose of the arrow syntax. There are no values that directly represent *signals* in Euterpea—there are only signal *functions*.

The arrow syntax provides a convenient way to compose signal functions together—i.e. to wire together the boxes that make up a signal processing diagram. By not giving the user direct access to signals, and providing a disciplined way to compose signal functions (namely arrow syntax), time- and space-leaks are avoided. In fact, the resulting framwework is highly amenable to optimization, although this requires using special features in Haskell, as described in Chapter **??**.

A signal function whose type is of the form $Clock\ c \Rightarrow SigFun\ c\ ()\ b$ essentially takes no input, but produces some output of type b. Because of this we often refer to such a signal function as a *signal source*.

19.1.2 Four Useful Functions

There are four useful auxiliary functions that will make writing signal functions a bit easier. The first two essentially "lift" constants and functions from the Haskell level to the arrow (signal function) level:

$$\begin{aligned}
&arr &&:: Clock\ c \Rightarrow (a \to b) \to SigFun\ c\ a\ b \\
&constA &&:: Clock\ c \Rightarrow \qquad\ \ b\ \ \to SigFun\ c\ ()\ b
\end{aligned}$$

For example, a signal function that adds one to every sample of its input can be written simply as $arr\ (+1)$, and a signal function that returns the constant 440 as its result can be written $constA\ 440$ (and is a signal source, as defined earlier).

The other two functions allow us to *compose* signal functions:

$$\begin{aligned}
&(\ggg) :: Clock\ clk \Rightarrow \\
&\qquad\quad SigFun\ clk\ a\ b \to SigFun\ clk\ b\ c \to SigFun\ clk\ a\ c \\
&(\lll) :: Clock\ clk \Rightarrow \\
&\qquad\quad SigFun\ clk\ b\ c \to SigFun\ clk\ a\ b \to SigFun\ clk\ a\ c
\end{aligned}$$

$(\lll)$ is analogous to Haskell's standard composition operator $(\circ)$, whereas $(\ggg)$ is like "reverse composition."

As an example that combines both of the ideas above, recall the very first example given in this chapter:

proc $y \to$ **do**

$$
\begin{aligned}
&x \leftarrow \mathit{sigfun} \prec y + 1 \\
&\mathit{outA} \prec x
\end{aligned}
$$

which essentially applies *sigfun* to one plus the input. This signal function can be written more succinctly as either *arr* (+1) ⋙ *sigfun* or *sigfun* ⋘ *arr* (+1).

The functions (⋙), (⋘), and *arr* are actually generic operators on arrows, and are defined in Haskell's *Arrow* library. Euterpea imports them from there and adds them to the Euterpea namespace, so they do not have to be explicitly imported by the user.

19.1.3 Some Simple Examples

Let's now work through a few examples that focus on the behavior of signal functions, so that we can get a feel for how they are used in practice. Euterpea has many pre-defined signal functions, including ones for sine waves, numeric computations, transcendental functions, delay lines, filtering, noise generation, integration, and so on. Many of these signal functions are inspired by csound [Ver86], where they are called *unit generators*. Some of them are not signal functions *per se*, but take a few fixed arguments to yield a signal function, and it is important to understand this distinction.

For example, there are several pre-defined functions for generating sine waves and periodic waveforms in Euterpea. Collectively these are called *oscillators*, a name taken from electronic circuit design. They are summarized in Figure 19.1.

The two most common oscillators in Euterpea are:

$$
\begin{aligned}
&\mathit{osc} &&:: \mathit{Clock}\ c \Rightarrow \\
& && \mathit{Table} \rightarrow \mathit{Double} \rightarrow \mathit{SigFun}\ c\ \mathit{Double}\ \mathit{Double} \\
&\mathit{oscFixed} &&:: \mathit{Clock}\ c \Rightarrow \\
& && \mathit{Double} \rightarrow \mathit{SigFun}\ c\ ()\ \mathit{Double}
\end{aligned}
$$

osc uses fixed-waveform table-lookup synthesis as described in Section 18.2.2. The first argument is the fixed wavetable; we will see shortly how such a table can be generated. The second argument is the initial phase angle, represented as a fraction between 0 and 1. The resulting signal function then converts a signal representing the desired output frequency to a signal that has that output frequency.

oscFixed uses an efficient recurrence relation to compute a pure sinusoidal wave; the mathematics of this are described in Section **??**. In contrast with *osc*, its single argument is the desired output frequency. The resulting

$$
\begin{aligned}
&osc, oscI :: Clock\ c \Rightarrow \\
&\qquad Table \rightarrow Double \rightarrow SigFun\ c\ Double\ Double
\end{aligned}
$$

osc tab ph is a signal function whose input is a frequency, and output is a signal having that frequency. The output is generated using fixed-waveform table-lookup, using the table *tab*, starting with initial offset (phase angle) *ph* expressed as a fraction of a cycle (0 to 1). *oscI* is the same, but uses linear interpolation between points.

$$
\begin{aligned}
&oscFixed :: Clock\ c \Rightarrow \\
&\qquad Double \rightarrow SigFun\ c\ ()\ Double
\end{aligned}
$$

oscFixed freq is a signal source whose sinusoidal output frequency is *freq*. It uses a recurrence relation that requires only one multiply and two add operations for each sample of output.

$$
\begin{aligned}
&oscDur, oscDurI :: Clock\ c \Rightarrow \\
&\qquad Table \rightarrow Double \rightarrow Double \rightarrow SigFun\ ()\ Double
\end{aligned}
$$

oscDur tab del dur samples just once through the table *tab* at a rate determined by *dur*. For the first *del* seconds, the point of scan will reside at the first location of the table; it will then move through the table at a constant rate, reaching the end in another *dur* seconds; from that time on (i.e. after $del + dur$ seconds) it will remain pointing at the last location. *oscDurI* is similar but uses linear interpolation between points.

$$
\begin{aligned}
&oscPartials :: Clock\ c \Rightarrow \\
&\qquad Table \rightarrow Double \rightarrow SigFun\ c\ (Double, Int)\ Double
\end{aligned}
$$

oscPartials tab ph is a signal function whose pair of inputs determines the frequency (as with *osc*), as well as the number of harmonics of that frequency, of the output. *tab* is the table that is cycled through, and *ph* is the phase angle (as with *osc*).

Figure 19.1: Euterpea's Oscillators

signal function is therefore a signal source (i.e. its input type is ()).

The key point here is that the frequency that is output by *osc* is an *input to the signal function*, and therefore can vary with time, whereas the frequency output by *oscFixed* is a *fixed argument*, and cannot vary with time. To see this concretely, let's define a signal source that generates a pure sine wave using *oscFixed* at a fixed frequency, say 440 Hz:

$$
\begin{aligned}
&s_1 :: \mathit{Clock}\ c \Rightarrow \mathit{SigFun}\ c\ ()\ \mathit{Double}\\
&s_1 = \mathbf{proc}\ () \rightarrow \mathbf{do}\\
&\quad s \leftarrow \mathit{oscFixed}\ 440 \prec ()\\
&\quad \mathit{outA} \prec s
\end{aligned}
$$

Since the resulting signal s is directly returned through *outA*, this example can also be written:

$$
\begin{aligned}
&s_1 = \mathbf{proc}\ () \rightarrow \mathbf{do}\\
&\quad \mathit{oscFixed}\ 440 \prec ()
\end{aligned}
$$

Alternatively, we could simply write *oscFixed* 440.

To use *osc* instead, we first need to generate a wavetable that represents one full cycle of a sine wave. We can do this using one of Eutperpea's table generating functions, which are summarized in Figure 19.2. For example, using Euterpea's *tableSinesN* function, we can define:

$$
\begin{aligned}
&\mathit{tab}_1 :: \mathit{Table}\\
&\mathit{tab}_1 = \mathit{tableSinesN}\ 4096\ [1]
\end{aligned}
$$

This will generate a table of 4096 elements, consisting of one sine wave whose peak amplitude is 1.0. Then we can define the following signal source:

$$
\begin{aligned}
&s_2 :: \mathit{Clock}\ c \Rightarrow \mathit{SigFun}\ c\ ()\ \mathit{Double}\\
&s_2 = \mathbf{proc}\ () \rightarrow \mathbf{do}\\
&\quad \mathit{osc}\ \mathit{tab}_1\ 0 \prec 440
\end{aligned}
$$

Alternatively, we could use the *const* and composition operators to write either $\mathit{constA}\ 440 \ggg \mathit{osc}\ \mathit{tab}_1\ 0$ or $\mathit{osc}\ \mathit{tab}_2\ 0 \lll \mathit{constA}\ 440$. s_1 and s_2 should be compared closely.

Keep in mind that *oscFixed* only generates a sine wave, whereas *osc* generates whatever is stored in the wavetable. Indeed, *tableSinesN* actually creates a table that is the sum of a series of overtones, i.e. multiples of the fundmental frequency (recall the discussion in Section 18.1.3). For example:

$$\mathit{tab}_2 = \mathit{tableSinesN}\ 4096\ [1.0, 0.5, 0.33]$$

generates a waveform consisting of the fundamental frequency with amplitude 1.0, the first overtone at amplitude 0.5, and the second overtone at

$$
\begin{array}{ll}
\textbf{type}\ \mathit{TableSize} & = \mathit{Int} \\
\textbf{type}\ \mathit{PartialNum} & = \mathit{Double} \\
\textbf{type}\ \mathit{PartialStrength} & = \mathit{Double} \\
\textbf{type}\ \mathit{PhaseOffset} & = \mathit{Double} \\
\textbf{type}\ \mathit{StartPt} & = \mathit{Double} \\
\textbf{type}\ \mathit{SegLength} & = \mathit{Double} \\
\textbf{type}\ \mathit{EndPt} & = \mathit{Double}
\end{array}
$$

$$
\begin{array}{l}
\mathit{tableLinear}, \mathit{tableLinearN} :: \\
\quad \mathit{TableSize} \rightarrow \mathit{StartPt} \rightarrow [(\mathit{SegLength}, \mathit{EndPt})] \rightarrow \mathit{Table}
\end{array}
$$

tableLinear size sp pts is a table of size *size* whose starting point is $(0, sp)$ and that uses straight lines to move from that point to, successively, each of the points in *pts*, which are segment-length/endpoint pairs (segment lengths are projections along the x-axis). *tableLinearN* is a normalized version of the result.

$$
\begin{array}{l}
\mathit{tableExpon}, \mathit{tableExponN} :: \\
\quad \mathit{TableSize} \rightarrow \mathit{StartPt} \rightarrow [(\mathit{SegLength}, \mathit{EndPt})] \rightarrow \mathit{Table}
\end{array}
$$

Just like *tableLinear* and *tableLinearN*, respectively, except that exponential curves are used to connect the points.

$$
\begin{array}{l}
\mathit{tableSines3}, \mathit{tableSines3N} :: \\
\quad \mathit{TableSize} \rightarrow [(\mathit{PartialNum}, \mathit{PartialStrength}, \mathit{PhaseOffset})] \rightarrow \mathit{Table}
\end{array}
$$

tableSines3 size triples is a table of size *size* that represents a sinusoidal wave and an arbitrary number of partials, whose relationship to the fundamental frequency, amplitude, and phase are determined by each of the triples in *triples*. *tableSines3N* is a normalized version of the result.

$$
\begin{array}{l}
\mathit{tableSines}, \mathit{tableSinesN} :: \\
\quad \mathit{TableSize} \rightarrow [\mathit{PartialStrength}] \rightarrow \mathit{Table}
\end{array}
$$

Like *tableSines3* and *tableSines3N*, respectively, except that the second argument is an ordered list of the strengths of each partial, starting with the fundamental.

$$
\begin{array}{l}
\mathit{tableBesselN} :: \\
\quad \mathit{TableSize} \rightarrow \mathit{Double} \rightarrow \mathit{Table}
\end{array}
$$

tableBesselN size x is a table representing the log of a modified Bessel function of the second kind, order 0, suitable for use in amplitude-modulated FM. x is the x-interval (0 to x) over which the function is defined.

Figure 19.2: Table Generating Functions

amplitude 0.33. So a more complex sound can be synthesized just by changing the wavetable:

$$
\begin{array}{l}
s_3 :: \mathit{Clock}\ c \Rightarrow \mathit{SigFun}\ c\ ()\ \mathit{Double} \\
s_3 = \mathbf{proc}\ () \rightarrow \mathbf{do} \\
\quad \mathit{osc}\ \mathit{tab}_2\ 0 \prec 440
\end{array}
$$

To get the same effect using *oscFixed* we would have to write:

$$
\begin{array}{l}
s_4 :: \mathit{Clock}\ c \Rightarrow \mathit{SigFun}\ c\ ()\ \mathit{Double} \\
s_4 = \mathbf{proc}\ () \rightarrow \mathbf{do} \\
\quad f_0 \leftarrow \mathit{oscFixed}\ 440 \ \ \prec () \\
\quad f_1 \leftarrow \mathit{oscFixed}\ 880 \ \ \prec () \\
\quad f_2 \leftarrow \mathit{oscFixed}\ 1320 \prec () \\
\quad \mathit{outA} \prec (f_0 + 0.5 * f_1 + 0.33 * f_2) / 1.83
\end{array}
$$

Not only is this more complex, it is less efficient. (The division by 1.83 is to normalize the result—if the peaks of the three signals f_0, f_1, and f_2 align properly, the peak amplitude will be 1.83 (or -1.83), which is outside the range ± 1.0 and may cause clipping (see discussion in Section 19.2).

So far in these examples we have generated a signal whose fundamental frequency is 440 Hz. But as mentioned, in the case of *osc*, the input to the oscillator is a signal, and can therefore itself be time-varying. As an example of this idea, let's implement *vibrato*—the performance effect whereby a musician slightly varies the frequency of a note in a pulsating rhythm. On a string instrument this is typically achieved by wiggling the finger on the fingerboard, on a reed instrument by an adjustment of the breath and emboucher to compress and relax the reed in a suitable way, and so on.

Specifically, let's define a function:

$$
\begin{array}{l}
\mathit{vibrato} :: \mathit{Clock}\ c \Rightarrow \\
\qquad\qquad \mathit{Double} \rightarrow \mathit{Double} \rightarrow \mathit{SigFun}\ c\ \mathit{Double}\ \mathit{Double}
\end{array}
$$

such that *vibrato* $f\ d$ is a signal function that takes a frequency argument (this is not a signal of a given frequency, it is the frequency itself), and generates a signal at that frequency, but with vibrato added, where f is the vibrato frequency, and d is the vibrato depth. We will consider "depth" to be a measure of how many Hz the input frequency is modulated.

Intuitively, it seems as if we need *two* oscillators, one to generate the fundamental frequency of interest, and the other to generate the vibrato (much lower in frequency). Here is a solution:

$$
\begin{array}{l}
\mathit{vibrato} :: \mathit{Clock}\ c \Rightarrow \\
\qquad\qquad \mathit{Double} \rightarrow \mathit{Double} \rightarrow \mathit{SigFun}\ c\ \mathit{Double}\ \mathit{Double}
\end{array}
$$

$$
\begin{array}{l}
\mathit{vibrato}\ \mathit{vfrq}\ \mathit{dep} = \mathbf{proc}\ \mathit{afrq} \rightarrow \mathbf{do} \\
\quad \mathit{vib}\ \ \leftarrow \mathit{osc}\ \mathit{tab}_1\ 0 \prec \mathit{vfrq} \\
\quad \mathit{aud} \leftarrow \mathit{osc}\ \mathit{tab}_2\ 0 \prec \mathit{afrq} + \mathit{vib} * \mathit{dep} \\
\quad \mathit{outA} \prec \mathit{aud}
\end{array}
$$

Note that a pure sine wave is used for the vibrato signal, whereas tab_2, a sum of three sine waves, is chosen for the signal itself.

For example, to play a 1000 Hz tone with a vibrato frequency of 5 Hz and a depth of 20 Hz, we could write:

$$
\begin{array}{l}
s_5 :: \mathit{AudSF}\ ()\ \mathit{Double} \\
s_5 = \mathit{constA}\ 1000 \ggg \mathit{vibrato}\ 5\ 20
\end{array}
$$

Vibrato is actually an example of a more general sound synthesis technique called *frequency modulation* (since one signal is being used to vary, or modulate, the frequency of another signal), and will be explained in more detail in Chapter **??**. Other chapters include synthesis techniques such as additive and subtractive synthesis, plucked instruments using waveguides, physical modeling, granular synthesis, as well as audio processing techniques such as filter design, reverb, and other effects. Now that we have a basic understanding of signal functions, these techniques will be straighforward to express in Euterpea.

19.2 Generating Sound

Euterpea can execute some programs in real-time, but sufficiently complex programs require writing the result to a file. The function for achieving this is:

$$
\begin{array}{l}
\mathit{outFile} :: (\mathit{AudioSample}\ a, \mathit{Clock}\ c) \Rightarrow \\
\qquad\qquad \mathit{String} \rightarrow \mathit{Double} \rightarrow \mathit{SigFun}\ c\ ()\ a \rightarrow \mathit{IO}\ ()
\end{array}
$$

The first argument is the name of the WAV file to which the result is written. The second argument is the duration of the result, in seconds (remember that signals are conceptually infinite). The third argument is a signal function that takes no input and generates a signal of type a as output (i.e. a signal source), where a is required to be an instance of the *AudioSample* type class, which allows one to choose between mono, stereo, etc.

For convenience, Euterpea defines these type synonyms:

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Mono}\ p = \mathit{SigFun}\ p\ ()\ \mathit{Double} \\
\mathbf{type}\ \mathit{Stereo}\ p = \mathit{SigFun}\ p\ ()\ (\mathit{Double}, \mathit{Double})
\end{array}
$$

For example, the IO command *outfile* `"test.wav"` 5 *sf* generates 5 seconds of output from the signal function *sf*, and writes the result to the file `"test.wav"`. If *sf* has type *Mono AudRate* (i.e. *SigFun AudRate* () *Double* then the result will be monophonic; if the type is *Stereo AudRate* (i.e. *SigFun AudRate* () (*Double*, *Double*) the result will be stereophonic.

One might think that *outFile* should be restricted to *AudRate*. However, by allowing a signal of any clock rate to be written to a file, one can use external tools to analyze the result of control signals or other signals of interest as well.

An important detail in writing WAV files with *outFile* is that care must be taken to ensure that each sample falls in the range ± 1.0. If this range is exceeded, the output sound will be harshly distorted, a phenomenon known as *clipping*. The reason that clipping sounds especially bad is that once the maximum limit is exceeded, the subsequent samples are interpreted as the *negation* of their intended value—and thus the signal swings abruptly from its largest possible value to its smallest possible value. Of course, signals within your program may be well outside this range—it is only when you are ready to write the result to a file that clipping needs to be avoided.

One can easily write signal functions that deal with clipping in one way or another. For example here's one that simply returns the maximum (positive) or mininum (negative) value if they are exceeded, thus avoiding the abrupt change in magnitude described above, and degenerating in the worst case to a square wave:

```
simpleClip :: Clock c => SigFun c Double Double
simpleClip = arr f where
  f x = if abs x <= 1.0 then x else signum x
```

> **Details:** *abs* is the absolute value function in Haskell, and *signum* returns -1 for negative numbers, 0 for zero, and 1 for positive numbers.

19.3 Instruments

So far we have only considered signal functions as stand-alone values whose output we can write to a WAV file. But how do we connect the ideas in previous chapters about *Music* values, *Performance*s, and so on, to the ideas presented in this chapter? This section presents a bridge between the two worlds.

19.3.1 Turning a Signal Function into an Instrument

Suppose that we have a *Music* value that, previously, we would have played using a MIDI instrument, and now we want to play using an instrument that we have designed using signal functions. To do this, first recall from Chapter 2 that the *InstrumentName* data type has a special constructor called *Custom*:

$$
\begin{aligned}
&\textbf{data}\ \mathit{InstrumentName} = \\
&\quad\ \ \mathit{AcousticGrandPiano} \\
&\quad \mid \mathit{BrightAcousticPiano} \\
&\quad \mid \ldots \\
&\quad \mid \mathit{Custom}\ \mathit{String} \\
&\quad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}, \mathit{Ord})
\end{aligned}
$$

With this constructor, names (represented as strings) can be given to instruments that we have designed using signal functions. For example:

$$
\begin{aligned}
&\mathit{simpleInstr} :: \mathit{InstrumentName} \\
&\mathit{simpleInstr} = \mathit{Custom}\ \texttt{"Simple Instrument"}
\end{aligned}
$$

Now we need to define the instrument itself. Euterpea defines the following type synonym:

$$\textbf{type}\ \mathit{Instr}\ a = \mathit{Dur} \to \mathit{AbsPitch} \to \mathit{Volume} \to [\mathit{Double}] \to a$$

Although *Instr* is polymorphic, by far its most common instantiation is the type *Instr* (*AufSF* () *Double*). An instrument of this type is a function that takes a duration, absolute pitch, volume, and a list of parameters, and returns a signal source that generates the resulting sound.

The list of parameters (similar to the "pfields" in csound) are not used by MIDI instruments, and thus have not been discussed until now. They afford us unlimited expressiveness in controlling the sound of our signal-function based instruments. Recall from Chapter 8 the types:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Music1} = \mathit{Music}\ \mathit{Note1} \\
&\textbf{type}\ \mathit{Note1}\ \ = (\mathit{Pitch}, [\mathit{NoteAttribute}])
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data}\ \mathit{NoteAttribute} = \\
&\quad\ \ \mathit{Volume}\ \mathit{Int} \\
&\quad \mid \mathit{Fingering}\ \mathit{Integer} \\
&\quad \mid \mathit{Dynamics}\ \mathit{String} \\
&\quad \mid \mathit{Params}\ [\mathit{Double}]
\end{aligned}
$$

$\quad$ **deriving** $(Eq, Show)$

Using the *Params* constructor, each individual note in a *Music1* value can be given a different list of parameters. It is up to the instrument designer to decide how these parameters are used.

There are three steps to playing a *Music* value using a user-defined instrument. First, we must coerce our signal function into an instrument having the proper type *Instr* as described above. For example, let's turn the *vibrato* function from the last section into a (rather primitive) instrument:

$$
\begin{array}{l}
myInstr :: Instr\ (AudSF\ ()\ Double) \\
\quad \texttt{--}\ Dur \rightarrow AbsPitch \rightarrow Volume \rightarrow [Double] \rightarrow (AudSF\ ()\ Double) \\
myInstr\ dur\ ap\ vol\ [vfrq, dep] = \\
\quad \mathbf{proc}\ () \rightarrow \mathbf{do} \\
\qquad vib\ \ \leftarrow osc\ tab_1\ 0 \prec vfrq \\
\qquad aud \leftarrow osc\ tab_2\ 0 \prec apToHz\ ap + vib * dep \\
\qquad outA \prec aud
\end{array}
$$

Aside from the re-shuffling of arguments, note the use of the function *apToHz*, which converts an absolute pitch into its corresponding frequency:

$$apToHz :: Floating\ a \Rightarrow AbsPitch \rightarrow a$$

Next, we must connect our instrument name (used in the *Music* value) to the instrument itself (such as defined above). This is achieved using a simple association list, or *instrument map*:

$$\mathbf{type}\ InstrMap\ a = [(InstrumentName, Instr\ a)]$$

Continuing the example started above:

$$
\begin{array}{l}
myInstrMap :: InstrMap\ (AudSF\ ()\ Double) \\
myInstrMap = [(simpleInstr, myInstr)]
\end{array}
$$

Finally, we need a function that is analogous to *perform* from Chapter 8, except that instead of generating a *Performance*, it creates a single signal function that will "play" our *Music* value for us. In Euterpea that function is called *renderSF*:

$$
\begin{array}{ll}
renderSF :: & (Performable\ a, AudioSample\ b, Clock\ c) \Rightarrow \\
& Music\ a \rightarrow \\
& InstrMap\ (SigFun\ p\ ()\ b) \rightarrow \\
& (Double, SigFun\ p\ ()\ b)
\end{array}
$$

The first element of the pair that is returned is the duration of the *Music* value, just as is returned by *perform*. That way we know how much of the signal function to render in order to hear the entire composition.

$$
\begin{array}{l}
mel :: Music1 \\
mel = \\
\quad \textbf{let}\ m = Euterpea.line\ [na_1\ (c\ 4\ en), na_1\ (ef\ 4\ en), na_1\ (f\ 4\ en), \\
\qquad\qquad na_2\ (af\ 4\ qn), na_1\ (f\ 4\ en),\ \ na_1\ (af\ 4\ en), \\
\qquad\qquad na_2\ (bf\ 4\ qn), na_1\ (af\ 4\ en), na_1\ (bf\ 4\ en), \\
\qquad\qquad na_1\ (c\ 5\ en),\ \ na_1\ (ef\ 5\ en), na_1\ (f\ 5\ en), \\
\qquad\qquad na_3\ (af\ 5\ wn)] \\
\qquad\quad na_1\ (Prim\ (Note\ d\ p)) = Prim\ (Note\ d\ (p, [Params\ [0, 0]])) \\
\qquad\quad na_2\ (Prim\ (Note\ d\ p)) = Prim\ (Note\ d\ (p, [Params\ [5, 10]])) \\
\qquad\quad na_3\ (Prim\ (Note\ d\ p)) = Prim\ (Note\ d\ (p, [Params\ [5, 20]])) \\
\quad \textbf{in}\ instrument\ simpleInstr\ m
\end{array}
$$

Figure 19.3: A Simple Melody

Using the simple melody *mel* in Figure 19.3, and the simple vibrato instrument defined above, we can generate our result and write it to a file, as follows:

$$
\begin{array}{ll}
(dr, sf) & = renderSF\ mel\ myInstrMap \\
main & = outFile\ \texttt{"simple.wav"}\ dr\ sf
\end{array}
$$

For clarity we show in Figure 19.4 all of the pieces of this running example as one program.

19.3.2 Envelopes

Most instruments played by humans have a distinctive sound that is partially dependent on how the performer plays a particular note. For example, when a wind instrument is played (whether it be a flute, saxophone, or trumpet), the note does not begin instantaneously—it depends on how quickly and forcibly the performer blows into the instrument. This is called the "attack." Indeed, it is not uncommon for the initial pulse of energy to generate a sound that is louder than the "sustained" portion of the sound. And when the note ends, the airflow does not stop instantaneously, so there is variability in the "release" of the note.

The overall variability in the loudness of a note can be simulated by

```haskell
simpleInstr :: InstrumentName
simpleInstr = Custom "Simple Instrument"

myInstr :: Instr (AudSF () Double)
myInstr dur ap vol [vfrq, dep] =
  proc () -> do
    vib <- osc tab1 0 -< vfrq
    aud <- osc tab2 0 -< apToHz ap + vib * dep
    outA -< aud

myInstrMap :: InstrMap (AudSF () Double)
myInstrMap = [(simpleInstr, myInstr)]

(d, sf) = renderSF mel myInstrMap
main    = outFile "simple.wav" d sf
```

Figure 19.4: A Complete Example of a Signal-Function Based Instrument

...

Figure 19.5: ADSR Envelope

multiplying the output of a signal function by an *envelope*, which is a time-varying signal that captures the desired behavior. Indeed, the *ADSR envelope* (attack, decay, sustain, release) introduced above is one of the most common envelopes used in practice. It is shown pictorially in Figure 19.5. Before defining it in Euterpea, however, we first describe a collection of simpler envelopes.

Figure 19.6 shows six pre-defined envelope-generating functions. Read the code comments carefully to understand what they do.

Here are some additional comments regarding *envCSEnvplx*, easily the most sophisticated of the envelope generators:

1. The fifth argument to *envCSEnvplx*: A value greater than 1 causes exponential growth; a value less than 1 causes exponential decay; a value = 1 will maintain a true steady state at the last rise value. The attenuation is not by a fixed rate (as in a piano), but is sensitive to a note's duration. However, if this argument is less than 0 (or if steady state is less than 4 k-periods) a fixed attenuation rate of *abs atss* per

```
-- a linear envelope
envLine      :: Clock p =>
  Double  ->  -- starting value
  Double  ->  -- duration in seconds
  Double  ->  -- value after dur seconds
  SigFun p () Double

-- an exponential envelope
envExpon     :: Clock p =>
  Double  ->  -- starting value; zero is illegal for exponentials
  Double  ->  -- duration in seconds
  Double  ->  -- value after dur seconds (must be non-zero
              -- and agree in sign with first argument)
  SigFun p () Double

-- a series of linear envelopes
envLineSeg   :: Clock p =>
  [Double] -> -- list of points to trace through
  [Double] -> -- list of durations for each line segment
              -- (one element fewer than previous argument)
  SigFun p () Double

-- a series of exponential envelopes
envExponSeg :: Clock p =>
  [Double] -> -- list of points to trace through
  [Double] -> -- list of durations for each line segment
              -- (one element fewer than previous argument)
  SigFun p () Double

-- an "attack/decay/release" envelope; each segment is linear
envASR       :: Clock p =>
  Double  ->  -- rise time in seconds
  Double  ->  -- overall duration in seconds
  Double  ->  -- decay time in seconds
  SigFun p () Double

-- a more sophisticated ASR
envCSEnvlpx :: Clock p =>
  Double  ->  -- rise time in seconds
  Double  ->  -- overall duration in seconds
  Double  ->  -- decay time in seconds
  Table   ->  -- table representing rise shape
  Double  ->  -- attenuation factor, by which the last value
              -- of the envlpx rise is modified during the
              -- note's pseudo steady state
  Double  ->  -- attenuation factor by which the closing
              -- steady state value is reduced exponentially
              -- over the decay period
  SigFun p () Double
```

Figure 19.6: Envelopes

second is used. A value of 0 is illegal.

2. The sixth arg to *envCSEnvplx*: Must be positive and is normally of the order of 0.01. A large or excessively small value is apt to produce a cutoff that is not audible. Values less than or equal to 0 are disallowed.

Exercise 19.1 Using the Euterpea function *osc*, create a simple sinusoidal wave, but using different table sizes, and different frequencies, and see if you can hear the differences (report on what you hear). Use *outFile* to write your results to a file, and be sure to use a decent set of speakers or headphones.

Exercise 19.2 The *vibrato* function varies a signals frequency at a given rate and depth. Define an analogous function *tremolo* that varies the volume at a given rate and depth. However, in a sense, *tremolo* is a kind of envelope (infinite in duration), so define it as a signal source, with which you can then shape whatever signal you wish. Consider the "depth" to be the fractional change to the volume; that is, a value of 0 would result in no tremolo, a value of 0.1 would vary the amplitude from 0.9 to 1.1, and so on. Test your result.

Exercise 19.3 Define an ADSR ("attack/decay/sustain/release") envelope generator (i.e. a signal source) called *envADSR*, with type:

```
type DPair = (Double, Double)   -- pair of duration and amplitude
envADSR   :: DPair → DPair → DPair → Double → AudSF () Double
```

The three *DPair* arguments are the duration and amplitude of the attack, decay, and release "phases," respectively, of the envelope. The sustain phase should hold the last value of the decay phase. The fourth argument is the duration of the entire envelope, and thus the duration of the sustain phase should be that value minus the sum of the durations of the other three phases. (Hint: use Euterpeas *envLineSeg* function.) Test your result.

Exercise 19.4 Generate a signal that causes clipping, and listen to the result. Then use *simpleClip* to "clean it up" somewhat—can you hear the difference? Now write a more ambitious clipping function. In particular, one that uses some kind of non-linear reduction in the signal amplitude as it approaches plus or minus one (rather than abruptly "sticking" at plus or minus one, as in *simpleClip*).

Exercise 19.5 Define two instruments, each of type *Instr* (*AudSF* () *Double*). These can be as simple as you like, but each must take at least two *Params*. Define an *InstrMap* that uses these, and then use *renderSF* to "drive" your instruments from a *Music1* value. Test your result.

Chapter 20

Spectrum Analysis

```
{-# LANGUAGE Arrows #-}
module Euterpea.Music.Signal.SpectrumAnalysis where
import Euterpea
import Euterpea.Experimental (fftA)
import Data.Complex (Complex ((:+)), polar)
import Data.Maybe (listToMaybe, catMaybes)
```

There are many situations where it is desirable to take an existing sound signal—in particular one that is recorded by a microphone—and analyze it for its spectral content. If one can do this effectively, it is then possible (at least in theory) to recreate the original sound, or to create novel variations of it. The thepry behind this approach is based on *Fourier's Theorem*, which states that any periodic signal can be decomposed into a weighted sum of (a potentially infinite number of) sine waves. In this chapter we discuss the theory as well as the pragmatics for doing spectrum analysis in Euterpea.

20.1 Fourier's Theorem

A *periodic signal* is a signal that repeats itself infinitely often. Mathematically, a signal x is periodic if there exists a real number T such that for all integers n:

$$x(t) = x(t + nT)$$

T is called the *period*, which may be just a few microseconds, a few seconds, or perhaps days—the only thing that matters is that the signal repeats itself. Usually we want to find the smallest value of T that satisfies the above property. For example, a sine wave is surely periodic; indeed, recall from Section 18.1.1 that:

$$\sin(2\pi k + \theta) = \sin\theta$$

for any integer k. In this case, $T = 2\pi$, and it is the smallest value that satisfies this property.

But in what sense is, for example, a single musical note periodic? Indeed it is not, unless it is repeated infinitely often, which would not be very interesting musically. Yet something we would like to know is the spectral content of that single note, or even of a small portion of that note, within an entire composition. This is one of the practical problems that we will address later in the chapter.

Recall from Section 18.1.1 that a sine wave can be represented by: $x(t) = A\sin(\omega t + \phi)$, where A is the amplitude, ω is the radian frequency, and ϕ is the phase angle. Joseph Fourier, a french mathematician and physicist, showed the following result. Any periodic signal $x(t)$ with period T can be represented as:

$$x(t) = C_0 + \sum_{n=1}^{\infty} C_n \cos(\omega_0 n t + \phi_n) \tag{20.1}$$

This is called *Fourier's Theorem.* $\omega_0 = {}^{2\pi}/_{T}$ is called the *fundamental frequency.* Note that the frequency of each cosine wave in the series is an integer multiple of the fundamental frequency. The above equation is also called the *Fourier series* or *harmonic series* (related, but not to be confused with, the mathematical definition of harmonic series, which has the precise form $1 + {}^{1}/_{2} + {}^{1}/_{3} + {}^{1}/_{4} + \cdots$).

The trick, of course, is determining what the coefficients C_0, ..., C_n and phase angles ϕ_1, ..., ϕ_n are. Determining the above equation for a particular periodic signal is called *Fourier analysis*, and synthesizing a sound based on the above equation is called *Fourier synthesis.* Theoretically, at least, we should be able to use Fourier analysis to decompose a sound of interest into its composite sine waves, and then regenerate it by artificially generating those composite sine waves and adding them together (i.e. additive synthesis, to be described in Chapter 21). Of course, we also have to deal with the fact that the representation may involve an *infinite* number of composite signals.

As discussed somewhat in Chapter 18, many naturally occurring vibrations in nature—including the resonances of most musical instruments—are characterized as having a fundamental frequency (the perceived pitch) and some combination of multiples of that frequency, which are often called *harmonics*, *overtones* or *partials*. So Fourier's Theorem seems to be a good match for this musical application.

20.1.1 The Fourier Transform

When studying Fourier analysis, it is more convenient, mathematically, to use *complex exponentials*. We can relate working with complex exponentials back to sines and cosines using *Euler's Formula*:

$$\begin{aligned} e^{j\theta} &= cos(\theta) + jsin(\theta) \\ cos(\theta) &= \frac{1}{2}(e^{j\theta} + e^{-j\theta}) \\ sin(\theta) &= \frac{1}{2}(e^{j\theta} - e^{-j\theta}) \end{aligned}$$

For a periodic signal $x(t)$, which we consider to be a function of time, we denote its *Fourier transform* by $\hat{x}(f)$, which is a function of frequency. Each point in $\hat{x}$ is a complex number that represents the magnitude and phase of the frequency f's presence in $x(t)$. Using complex exponentials, the formula for $\hat{x}(f)$ in terms of $x(t)$ is:

$$\hat{x}(f) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt$$

where $\omega = 2\pi f$, and j is the same as the imaginary unit i used in mathematics.[1] Intuitively, the Fourier transform at a particular frequency f is the integral of the product of the original signal and a pure sinusiodal wave $e^{-j\omega t}$. This latter process is related to the *convolution* of the two signals, and intuitively will be non-zero only when the signal has some content of that pure signal in it.

The above equation describes $\hat{x}$ in terms of x. We can also go the other way around—defining x in terms of $\hat{x}$:

$$x(t) = \int_{-\infty}^{\infty} \hat{x}(f)e^{j\hat{\omega} f}df$$

[1] Historically, engineers prefer to use the symbol j rather than i, because i is generally used to represent current in an electrical circuit.

where $\hat{\omega} = 2\pi t$. This is called the *inverse* Fourier transform.

If we expand the definitions of ω and $\hat{\omega}$ we can see how similar these two equations are:

$$\hat{x}(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt \tag{20.2}$$

$$x(t) = \int_{-\infty}^{\infty} \hat{x}(f)e^{j2\pi ft}df \tag{20.3}$$

These two equations, for the Fourier transform and its inverse, are remarkable in their simplicity and power. They are also remarkable in the following sense: *no information is lost when converting from one to the other*. In other words, a signal can be represented in terms of its time-varying behavior or its spectral content—they are equivalent!

A function that has the property that $f(x) = f(-x)$ is called an *even* function; if $f(x) = -f(-x)$ it is said to be *odd*. It turns out that, perhaps surprisingly, *any* function can be expressed as the sum of a single even function and a single odd function. This may help provide some intuition about the equations for the Fourier transform, because the complex exponential $e^{j2\pi ft}$ separates the waveform by which it is being multiplied into its even and odd parts (recall Euler's formula). The real (cosine) part affects only the even part of the input, and the imaginary (sine) part affects only the odd part of the input.

20.1.2 Examples

Let's consider some examples, which are illustrated in Figure 20.1:

- Intuitively, the Fourier transform of a pure cosine wave should be an impulse function—that is, the spectral content of a cosine wave should be concentrated completely at the frequency of the cosine wave. The only catch is that, when working in the complex domain, the Fourier transform also yields the mirror image of the spectral content, at a frequency that is the negation of the cosine wave's frequency, as shown in Figure 20.1a. In other words, in this case, $\hat{x}(f) = \hat{x}(-f)$, i.e. $\hat{x}$ is even. So the spectral content is the *real* part of the complex number returned from the Fourier transform (recall Euler's formula).
- In the case of a pure sine wave, we should expect a similar result. The only catch now is that the spectral content is contained in the *imaginary* part of the complex number returned from the Fourier transform

(recall Euler's formula), and the mirror image is negated. That is, $\hat{x}(f) = -\hat{x}(-f)$, i.e. $\hat{x}$ is odd. This is illustrated in Figure 20.1b.

- Conversely, consider what the spectral content of an impulse function should be. Because an impulse function is infinitely "sharp," it would seem that its spectrum should contain energy at every point in the frequency domain. Indeed, the Fourier transform of an impulse function centered at zero is a constant, as shown in Figure 20.1c.

- Consider now the spectral content of a square wave. It can be shown that the Fourier series representation of a square wave is the sum of the square wave's fundamental frequency plus its harmonically decreasing (in magnitude) odd harmonics. Specifically:

$$sq(t) = \sum_{k=1}^{\infty} \frac{1}{k} \sin k\omega t, \quad \text{for odd } k \tag{20.4}$$

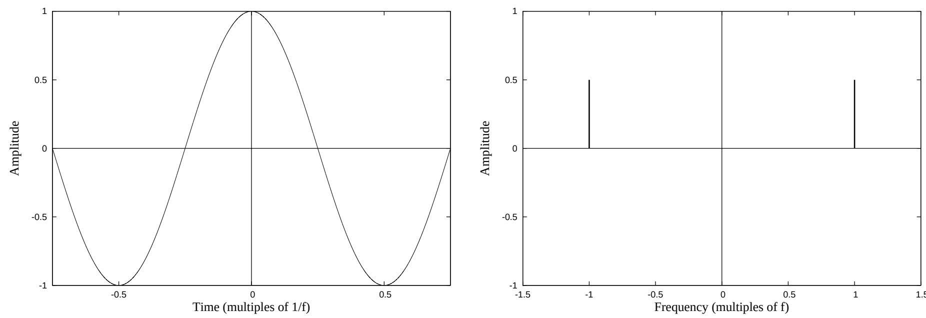
  The spectral content of this signal in shown in Figure 20.1d. Figure 20.2 also shows partial reconstruction of the square wave from a finite number of its composite signals.

It is worth noting that the diagrams in Figure 20.1 make no assumptions about time or frequency. Therefore, because the Fourier transform and its inverse are true mathematical inverses, we can read the diagrams as time domain / frequency domain pairs, or the other way around; i.e. as frequency domain / time domain pairs. For example, interpreting the diagram on the left of Figure 20.1a in the frequency domain, is to say that it is the Fourier transform of the signal on the right (interpreted in the time domain).

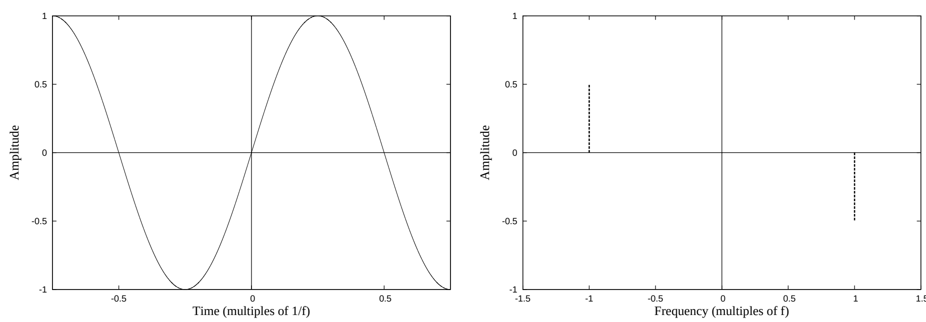
20.2 The Discrete Fourier Transform

Recall from Section 18.2.1 that we can move from the continuous signal domain to the discrete domain by replacing the time t with the quantity n/r, where n is the integer index into the sequence of discrete samples, and r is the sampling rate. Let us assume that we have done this for x, and we will use square brackets to denote the difference. That is, $x[n]$ denotes the n^{th} sample of the continuous signal $x(t)$, corresponding to the value $x(n/r)$.

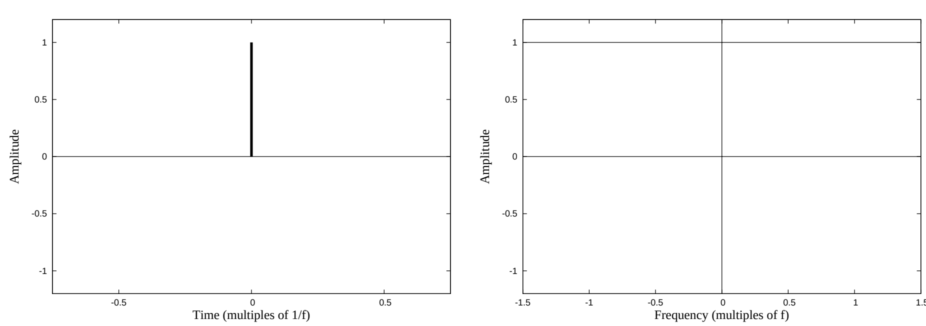
We would now like to compute the *Discrete Fourier Transform* (DFT) of our discrete signal. But instead of being concerned about the sampling rate (which can introduce aliasing, for example), our concern turns to the



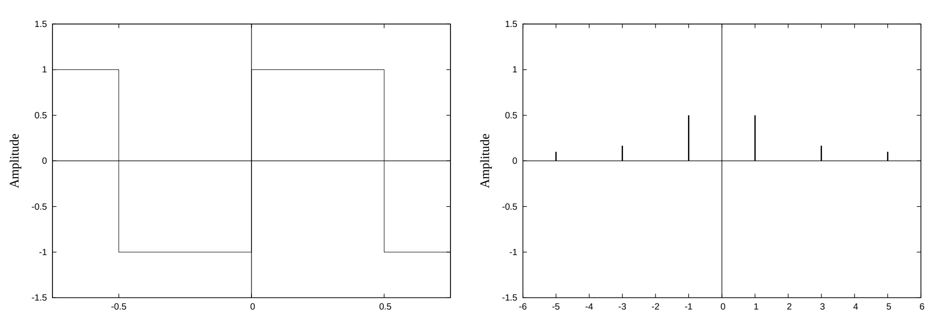
(a) Cosine wave



(b) Sine wave

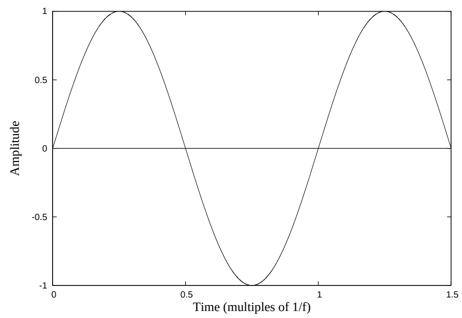


(c) Impulse function

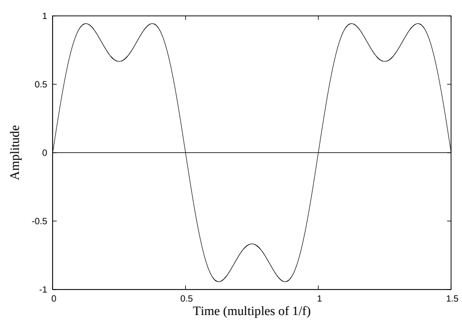


(d) Square wave

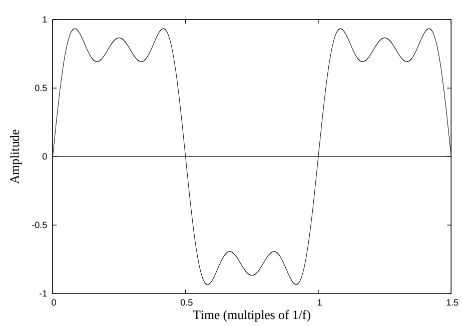
Figure 20.1: Examples of Fourier Transforms



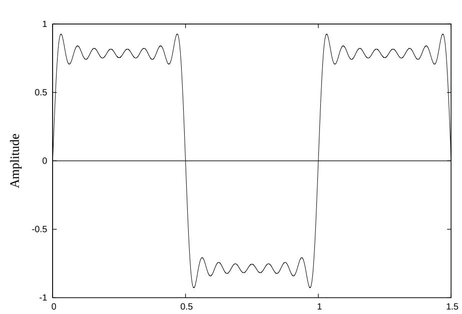
(a) Sine wave



(b) Sine wave + third harmonic



(c) Sine wave + third and fifth harmonics



(d) Sum of first eight terms of the Fourier series of a square wave

Figure 20.2: Generating a Square Wave from Odd Harmonics

number of samples that we use in computing the DFT—let's call this N. Intuitively, the integrals used in our equations for the Fourier transform and its inverse should become sums over the range $0...N-1$. This leads to a reformulation of our two equations (20.2 and 20.3) as follows:[2]

$$\hat{x}[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi kn}{N}}, \quad k = 0, 1, ..., N-1 \tag{20.5}$$

$$x[n] = \sum_{k=0}^{N-1} \hat{x}[k]e^{j\frac{2\pi kn}{N}}, \quad n = 0, 1, ..., N-1 \tag{20.6}$$

Despite all of the mathematics up to this point, the reader may now realize that the discrete Fourier transform as expressed above is amenable to implementation—for example it should not be difficult to write Haskell functions that realize each of the above equations. But before addressing implementation issues, let's discuss a bit more what the results actually *mean*.

20.2.1 Interpreting the Frequency Spectrum

Just as $x[n]$ represents a sampled version of the continuous input signal, $\hat{x}[k]$ represents a sampled version of the continous frequency spectrum. Care must be taken when interpreting either of these results, keeping in mind the Nyquist-Shannon Sampling Theorem (recall Section 18.2) and aliasing (Section 18.2.3).

Also recall that the result of a Fourier transform of a periodic signal is a Fourier series (see Section 20.1), in which the signal being analyzed is expressed as multiples of a fundamental frequency. In equation 20.5 above, that fundamental frequency is the inverse of the duration of the N samples, i.e. the inverse of N/r, or r/N. For example, if the sampling rate is 44.1 kHz (the CD standard), then:

- If we take $N = 441$ samples, then the fundamental frequency will be $r/N = 100$ Hz.

[2]The purpose of the factor $1/N$ in Equation 20.5 is to ensure that the DFT and the inverse DFT are in fact inverses of each other. But it is just by convention that one equation has this factor and the other does not—it would be sufficient if it were done the other way around. In fact, all that matters is that the product of the two coefficients be $1/N$, and thus it would also be sufficient for each equation to have the same coefficient, namely $1/\sqrt{N}$. Similarly, the negative exponent in one equation and positive in the other is also by convention—it would be sufficient to do it the other way around.

- If we take $N = 4410$ samples, then the fundamental frequency will be $r/N = 10$ Hz.
- If we take $N = 44100$ samples, then the fundamental frequency will be $r/N = 1$ Hz.

Thus, as would be expected, taking more samples yields a *finer* resolution of the frequency spectrum. On the other hand, note that if we increase the sampling rate and keep the number of samples fixed, we get a *coarser* resolution of the spectrum—this also should be expected, because if we increase the sampling rate we would expect to have to look at more samples to get the same accuracy.

Analogous to the Nyquist-Shannon Sampling Theorem, the representable points in the resulting frequency spectrum lie in the range $\pm r/2$, i.e. between plus and minus one-half of the sampling rate. For the above three cases, respectively, that means the points are:

- -22.0 kHz, -21.9 kHz, ..., -0.1 kHz, 0, 0.1 kHz, ..., 21.9 kHz, 22.0 kHz
- -22.05 kHz, -22.04 kHz, ..., -10 Hz, 0, 10 Hz, ..., 22.04 kHz, 22.05 kHz
- -22.05 kHz, -22.049 kHz, ..., -1 Hz, 0, 1 Hz, ..., 22.049 kHz, 22.05 kHz

For practical purposes, the first of these is usually too coarse, the third is too fine, and the middle one is useful for many applications.

Note that the first range of frequencies above does not quite cover the range $\pm r/2$. But remember that this is a discrete representation of the actual frequency spectrum, and the proper interpretation would include the frequences $+r/2$ and $-r/2$.

Also note that there are $N+1$ points in each of the above ranges, not N. Indeed, the more general question is, how do these points in the frequency spectrum correspond to the indices $i = 0, 1, ..., N-1$ in $\hat{x}[i]$? If we denote each of these frequencies as f, the answer is that:

$$f = \frac{ir}{N}, \qquad i = 0, 1, ..., N-1 \tag{20.7}$$

But note that this range of frequencies extends from 0 to $(N-1)(r/N)$, which exceeds the Nyquist-Shannon sampling limit of $r/2$. The way out of this dilemma is to realize that the DFT assumes that the input signal is periodic in time, and therefore the DFT is periodic in frequency. In other words, values of f for indices i greater than $^N/_2$ can be interpreted as frequencies that are the *negation* of the frequency given by the formula above. Assuming even N, we can revise formula 20.7 as follows:

$$f = \begin{cases} i\dfrac{r}{N}, & i = 0, 1, ..., \dfrac{N}{2} \\ (i-N)\dfrac{r}{N} & i = \dfrac{N}{2}, \dfrac{N}{2}+1, ..., N-1 \end{cases} \tag{20.8}$$

Note that when $i = {}^N/_2$, both equations apply, yielding $f = {}^r/_2$ in the first case, and $f = -{}^r/_2$ in the second. Indeed, the magnitude of the DFT for each of these frequencies is the same (see discussion in the next section), reflecting the periodicity of the DFT, and thus is simply a form of redundancy.

The above discussion has assumed a periodic signal whose fundamental frequency is known, thus allowing us to parameterize the DFT with the same fundamental frequency. In practice this rarely happens. That is, the fundamental frequency of the DFT typically has no integral relationship to the period of the periodic signal. This raises the question, what happens to the frequencies that "fall in the gaps" between the frequencies discussed above? The answer is that the energy of that frequency component will be distributed amongst neighboring points in a way that makes sense mathematically, although the result may look a little funny compared to the ideal result (where every frequency component is an integer multiple of the fundamental). The important thing to remember is that these are digital representations of the exact spectra, just as a digitized signal is representative of an exact signal. Two digitized signals can look very different (depending on sample rate, phase angle, and so on), yet represent the same underlying signal—the same is true of a digitized spectrum.

In practice, for reasons of computational efficiency, N is usually chosen to be a power of two. We will return to this issue when we discuss implementing the DFT.

20.2.2 Amplitude and Power of Spectrum

We discussed above how each sample in the result of a DFT relates to a point in the frequency spectrum of the input signal. But how do we determine the amplitude and phase angle of each of those frequency components? In general each sample in the result of a DFT is a complex number, thus having both a real and imaginary part, of the form $a + jb$. We can visualize this number as a point in the complex Cartesian plane, where the abscissa (x-axis) represents the real part, and the ordinate (y-axis) represents the imaginary part, as shown in Figure 20.3. It is easy to see that the line from the origin to the point of interest is a vector A, whose length is the *amplitude* of the frequency component in the spectrum:

$$A = \sqrt{a^2 + b^2} \tag{20.9}$$

The angle θ is the *phase*, and it is easily defined from the figure as:

$$\theta = tan^{-1}\frac{b}{a} \tag{20.10}$$

(This amplitude / phase pair is often called the *polar* representation of a complex number.)

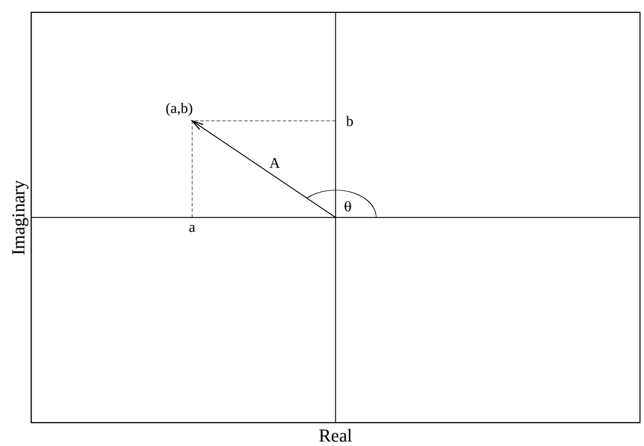


Figure 20.3: Complex and Polar Coordinates

Recall from Section 18.1.2 that power is proportional to the square of the amplitude. Since taking a square root adds computational expense, the square root is often omitted from Equation 20.9, thus yielding a *power spectrum* instead of an *amplitude spectrum*.

One subtle aspect of the resulting DFT is how to interpret *negative* frequencies. In the case of having an input whose samples are all real numbers (i.e. there are no imaginary components), which is true for audio applications, the negative spectrum is a mirror image of the positive spectrum, and the amplitude/power is distributed evenly between the two.

20.2.3 A Haskell Implementation of the DFT

From equation 20.5, which defines the DFT mathematically, we can write a Haskell program that implements the DFT.

The first thing we need to do is understand how complex numbers are handled in Haskell. They are captured in the *Complex* library, which must be imported into any program that uses them. The type *Complex T* is the type of complex numbers whose underlying numeric type is T. We will use, for example, *Complex Double* for testing our DFT. A complex number $a + jb$ is represented in Haskell as $a :\!+\ b$, and since $(:\!+)$ is a constructor, such values can be pattern matched.

Details: Complex numbers in Haskell are captured in the *Complex* library, in which complex numbers are defined as a polymorphic data type:

$$\textbf{infix}\ 6\ :\!+$$
$$\textbf{data}\ (\mathit{RealFloat}\ a) \Rightarrow \mathit{Complex}\ a = !a :\!+\ !a$$

The "!" in front of the type variables declares that the constructor $(:\!+)$ is strict in its arguments. For example, the complex number $a + jb$ is represented by $a :\!+\ b$ in Haskell. One can pattern match on complex number values to extract the real and imaginary parts, or use one of the predefined selectors defined in the *Complex* library:

$$\mathit{realPart}, \mathit{imagPart} :: \mathit{RealFloat}\ a \Rightarrow \mathit{Complex}\ a \rightarrow a$$

The *Complex* library also defines the following functions:

$$\begin{array}{ll}
\mathit{conjugate} & :: \mathit{RealFloat}\ a \Rightarrow \mathit{Complex}\ a \rightarrow \mathit{Complex}\ a \\
\mathit{mkPolar} & :: \mathit{RealFloat}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Complex}\ a \\
\mathit{cis} & :: \mathit{RealFloat}\ a \Rightarrow a \rightarrow \mathit{Complex}\ a \\
\mathit{polar} & :: \mathit{RealFloat}\ a \Rightarrow \mathit{Complex}\ a \rightarrow (a, a) \\
\mathit{magnitude}, \mathit{phase} & :: \mathit{RealFloat}\ a \Rightarrow \mathit{Complex}\ a \rightarrow a
\end{array}$$

The library also declares instances of *Complex* for the type classes *Num*, *Fractional*, and *Floating*.

Although not as efficient as arrays, for simplicity we choose to use lists to represent the vectors that are the input and output of the DFT. Thus if xs is the list that represents the signal x, then $xs\,!!\,n$ is the $n+1^{th}$ sample of that signal, and is equivalent to $x[n]$. Furthermore, using list comprehensions, we can make the Haskell code look very much like the mathematical definition captured in Equation 20.5. Finally, we adopt the convention that the length of the input signal is the number of samples that we will use for the DFT.

Probably the trickiest part of writing a Haskell program for the DFT is dealing with the types! In particular, if you look closely at Equation 20.5 you will see that N is used in three different ways—as an integer (for indexing), as a real number (in the exponent of e), and as a complex number (in the expression $^1/_N$).

Here is a Haskell program that implements the DFT:

```
dft :: RealFloat a => [Complex a] -> [Complex a]
dft xs =
  let lenI = length xs
      lenR = fromIntegral lenI
      lenC = lenR :+ 0
  in [let i = -2 * pi * fromIntegral k / lenR
      in (1 / lenC) * sum [(xs !! n) * exp (0 :+ i * fromIntegral n)
                          | n <- [0, 1 .. lenI - 1]]
     | k <- [0, 1 .. lenI - 1]]
```

Note that *lenI*, *lenR*, and *lenC* are the integer, real, and complex versions, respectively, of N. Otherwise the code is fairly straightforward—note in particular how list comprehensions are used to implement the ranges of n and k in Equation 20.5.

To test our program, let's first create a couple of waveforms. For example, recall that Equation 20.4 defines the Fourier series for a square wave. We can implement the first, first two, and first three terms of this series, corresponding respectively to Figures 20.2a, 20.2b, and 20.2c, by the following Haskell code:

```
mkTerm :: Int -> Double -> [Complex Double]
mkTerm num n = let f = 2 * pi / fromIntegral num
  in [sin (n * f * fromIntegral i) / n :+ 0
     | i <- [0, 1 .. num - 1]]

mkxa, mkxb, mkxc :: Int -> [Complex Double]
mkxa num = mkTerm num 1
mkxb num = zipWith (+) (mkxa num) (mkTerm num 3)
mkxc num = zipWith (+) (mkxb num) (mkTerm num 5)
```

Thus *mkTerm num n* is the n^{th} term in the series, using *num* samples.

Using the helper function *printComplexL* defined in Figure 20.4, which "pretty prints" a list of complex numbers, we can look at the result of our

```
printComplexL :: [Complex Double] -> IO ()
printComplexL xs =
  let f (i, rl :+ im) =
        do putStr (spaces (3 - length (show i)))
           putStr (show i ++ ":   ("          )
           putStr (niceNum rl  ++ ", "        )
           putStr (niceNum im ++ ")\n"        )
  in mapM_ f (zip [0 .. length xs - 1] xs)

niceNum :: Double -> String
niceNum d =
  let d' = fromIntegral (round (1e10 * d)) / 1e10
      (dec, fra)  = break (== '.') (show d')
      (fra', exp) = break (== 'e') fra
  in spaces (3 - length dec) ++ dec ++ take 11 fra'
      ++ exp ++ spaces (12 - length fra' - length exp)

spaces :: Int -> String
spaces n = take n (repeat ' ')
```

Figure 20.4: Helper Code for Pretty-Printing DFT Results

DFT in a more readable form.[3]

For example, suppose we want to take the DFT of a 16-sample representation of the first three terms of the square wave series. Typing the following at the GHCi prompt:

printComplexL (*dft* (*mkxc* 16))

will yield the result of the DFT, pretty-printing each number as a pair, along with its index:

```
  0:   (  0.0          ,   0.0          )
  1:   (  0.0          ,  -0.5          )
  2:   (  0.0          ,   0.0          )
  3:   (  0.0          ,  -0.1666666667 )
  4:   (  0.0          ,   0.0          )
  5:   (  0.0          ,  -0.1          )
  6:   (  0.0          ,   0.0          )
```

[3] "Pretty-printing" real numbers is a subtle task. The code in Figure 20.4 rounds the number to 10 decimal places of accuracy, and inserts spaces before and after to line up the decimal points and give a consistent string length. The fractional part is not padded with zeros, since that would give a false impression of its accuracy. (It is not necessary to understand this code in order to understand the concepts in this chapter.)

```
 7:  (  0.0           ,   0.0          )
 8:  (  0.0           ,   0.0          )
 9:  (  0.0           ,   0.0          )
10:  (  0.0           ,   0.0          )
11:  (  0.0           ,   0.1          )
12:  (  0.0           ,   0.0          )
13:  (  0.0           ,   0.1666666667 )
14:  (  0.0           ,   0.0          )
15:  (  0.0           ,   0.5          )
```

Let's study this result more closely. For sake of argument, assume a sample rate of 1.6 KHz. Then by construction using *mkxc*, our square-wave input's fundamental frequency is 100 Hz. Similarly, recall that the resolution of the DFT is r/N, which is also 100 Hz.

Now compare the overall result to Figure 20.1b. Recalling also Equation 20.8, we note that the above DFT results are non-zero precisely at 100, 300, 500, -500, -300, and -100 Hz. This is just what we would expect. Furthermore, the amplitudes are one-half of the corresponding harmonically decreasing weights dictated by Equation 20.4, namely the values 1, $^1/_6$, and $^1/_{10}$ (recall the discussion in Section 20.2.2).

Let's do another example. We can create an impulse function as follows:

$$\mathit{mkPulse} :: \mathit{Int} \rightarrow [\mathit{Complex\ Double}]$$
$$\mathit{mkPulse}\ n = 100 : \mathit{take}\ (n-1)\ (\mathit{repeat}\ 0)$$

and print its DFT with the command:

$$\mathit{printComplexL}\ (\mathit{dft}\ (\mathit{mkPulse}\ 16))$$

whose effect is:

```
 0:  (  6.25          ,   0.0          )
 1:  (  6.25          ,   0.0          )
 2:  (  6.25          ,   0.0          )
 3:  (  6.25          ,   0.0          )
 4:  (  6.25          ,   0.0          )
 5:  (  6.25          ,   0.0          )
 6:  (  6.25          ,   0.0          )
 7:  (  6.25          ,   0.0          )
 8:  (  6.25          ,   0.0          )
 9:  (  6.25          ,   0.0          )
10:  (  6.25          ,   0.0          )
11:  (  6.25          ,   0.0          )
12:  (  6.25          ,   0.0          )
13:  (  6.25          ,   0.0          )
```

```
14:  (  6.25          ,  0.0          )
15:  (  6.25          ,  0.0          )
```

Compare this to Figure 20.1c, and note how the original magnitude of the impulse (100) is distributed evenly among the 16 points in the DFT $(100/16 = 6.25)$.

So far we have considered only input signals whose frequency components are integral multiples of the DFT's resolution. This rarely happens in practice, however, because music is simply too complex, and noisy. As mentioned in 20.2.1, the energy of the signals that "fall in the gaps" is distributed among neighboring points, although not in as simple a way as you might think. To get some perspective on this, let's do one other example. We define a function to generate a signal whose frequeny is π times the fundamental frequency:

$$
\begin{aligned}
&x_1\ num = \mathbf{let}\ f = pi * 2 * pi / fromIntegral\ num \\
&\quad \mathbf{in}\ map\ (:+0)\ [sin\ (f * fromIntegral\ i) \\
&\qquad\qquad\qquad\quad \mid i \leftarrow [0, 1 \mathinner{..} num - 1]]
\end{aligned}
$$

π is an irrational number, but any number that "falls in the gaps" between indices would do. We can see the result by typing the command:

$$printComplexL\ (dft\ x_1)$$

which yields:

```
 0:  ( -7.9582433e-3 ,   0.0           )
 1:  ( -5.8639942e-3 ,  -1.56630897e-2)
 2:  (  4.7412105e-3 ,  -4.56112124e-2)
 3:  (  0.1860052232 ,  -0.4318552865 )
 4:  ( -5.72962095e-2,   7.33993364e-2)
 5:  ( -3.95845728e-2,   3.14378088e-2)
 6:  ( -3.47994673e-2,   1.65400768e-2)
 7:  ( -3.29813518e-2,   7.4048103e-3 )
 8:  ( -3.24834325e-2,   0.0           )
 9:  ( -3.29813518e-2,  -7.4048103e-3 )
10:  ( -3.47994673e-2,  -1.65400768e-2)
11:  ( -3.95845728e-2,  -3.14378088e-2)
12:  ( -5.72962095e-2,  -7.33993364e-2)
13:  (  0.1860052232 ,   0.4318552865 )
14:  (  4.7412105e-3 ,   4.56112124e-2)
15:  ( -5.8639942e-3 ,   1.56630897e-2)
```

This is much more complicated than the previous examples! Not only do the points in the spectrum seem to have varying amounts of energy, they also have both non-zero real and non-zero imaginary components, meaning that the magnitude and phase vary at each point. We can define a function that converts a list of complex numbers into a list of their polar representations as follows:

$$\begin{aligned}&\mathit{mkPolars} :: [\mathit{Complex\ Double}] \to [\mathit{Complex\ Double}]\\&\mathit{mkPolars} = \mathit{map}\ ((\lambda(m,p) \to m :\!+\ p) \circ \mathit{polar})\end{aligned}$$

which we can then use to reprint our result:

$$\mathit{printComplexL}\ (\mathit{mkPolars}\ (\mathit{dft}\ x_1))$$

```
 0:  (  7.9582433e-3 ,   3.1415926536 )
 1:  (  1.67247961e-2,  -1.9290259418 )
 2:  (  4.58569709e-2,  -1.4672199604 )
 3:  (  0.470209455  ,  -1.1640975898 )
 4:  (  9.31145435e-2,   2.2336013741 )
 5:  (  5.05497204e-2,   2.4704023271 )
 6:  (  3.85302097e-2,   2.6979021519 )
 7:  (  3.38023784e-2,   2.9207398294 )
 8:  (  3.24834325e-2,  -3.1415926536 )
 9:  (  3.38023784e-2,  -2.9207398294 )
10:  (  3.85302097e-2,  -2.6979021519 )
11:  (  5.05497204e-2,  -2.4704023271 )
12:  (  9.31145435e-2,  -2.2336013741 )
13:  (  0.470209455  ,   1.1640975898 )
14:  (  4.58569709e-2,   1.4672199604 )
15:  (  1.67247961e-2,   1.9290259418 )
```

If we focus on the magnitude (the first column), we can see that there is a peak near index 3 (corresponding roughly to the frequency π), with small amounts of energy elsewhere.

Exercise 20.1 Write a Haskell function *idft* that implements the *inverse* DFT as captured in Equation 20.3. Test your code by applying *idft* to one of the signals used earlier in this section. In other words, show empirically that, up to round-off errors, $\mathit{idft}\ (\mathit{dft}\ \mathit{xs}) == \mathit{xs}$.

Exercise 20.2 Use *dft* to analyze some of the signals generated using signal functions defined in Chapter 19.

Exercise 20.3 Define a function $\mathit{mkSqWave} :: \mathit{Int} \to \mathit{Int} \to [\mathit{Complex\ Double}]$ such that $\mathit{mkSqWave}\ \mathit{num}\ n$ is the sum of the first n terms of the Fourier series of a square wave, having *num* samples in the result.

Exercise 20.4 Prove mathematically that x and $\hat{x}$ are inverses. Also prove, using equational reasoning, that *dft* and *idft* are inverses. (For the latter you may assume that Haskell numeric types obey the standard axioms of real arithmetic.)

20.3 The Fast Fourier Transform

In the last section a DFT program was developed in Haskell that was easy to understand, being a faithful translation of Equation 20.5. For pedogogical purposes, this effort served us well. However, for practical purposes, the program is inherently inefficient.

To see why, think of $x[n]$ and $\hat{x}[k]$ as vectors. Thus, for example, each element of $\hat{x}$ is the sum of N multiplications of a vector by a complex exponential (which can be represented as a pair, the real and imaginary parts). And this overall process must be repeated for each value of k, also N times. Therefore the overall time complexity of the implied algorithm is $\mathrm{O}(N^2)$. For even moderate values of N, this can be computationally intractable. (Our choice of lists for the implementation of vectors makes the complexity even worse, because of the linear-time complexity of indexing, but the discussion below makes this a moot point.)

Fortunately, there exists a much faster algorithm called the *Fast Fourier Transform*, or FFT, that reduces the complexity to $\mathrm{O}(N \log N)$. This difference is quite significant for large values of N, and is the standard algorithm used in most signal processing applications. We will not go into the details of the FFT algorithm, other than to note that it is a divide-and-conquer algorithm that depends on the vector size being a power of two.[4]

Rather than developing our own program for the FFT, we will instead use the Haskell library *Numeric.FFT* to import a function that will do the job for us. Specifically:

fft :: ...

With this function we could explore the use of the FFT on specific iinput vectors, as we did earlier with *dft*.

However, our ultimate goal is to have a version of FFT that works on *signals*. We would like to be able to specify the number of samples as a power

[4]The basic FFT algorithm was invented by James Cooley and John Tukey in 1965.

of two (which we can think of as the "window size"), the clock rate, and how often we would like to take a snapshot of the current window (and thus successive windows may or may not overlap). The resulting signal function takes a signal as input, and outputs *events* at the specified rate. Events are discussed in more detail in Chapter 17.

Indeed, Euterpea provide this functionality for us in a function called *fftA*:

$$\mathit{fftA} :: \mathit{Int} \rightarrow \mathit{Double} \rightarrow \mathit{Int} \rightarrow \mathit{SF}\ \mathit{Double}\ (\mathit{Event}\ \mathit{FFTData})$$
$$\mathbf{type}\ \mathit{FFTData} = \mathit{Map}\ \mathit{Double}\ \mathit{Double}$$

SF is a signal function type similar to *SigFun*, except that it is targeted for use in the Musical User Interface (MUI) discussed in detail in Chapter **??**, and thus, for example, does not have a clock rate. *Map* T_1 T_2 is an abstract type that maps values of type T_1 to values of type T_2, and is imported from *Data.Map*.

fftA winInt rate size is a signal function that, every *winInt* samples of the input, creates a window of size 2ˆ*size*, and computes the FFT of that window. For every such result, it issues an *Event* that maps from frequency to magnitude (using the clock rate *rate* to determine the proper mapping).

Combining *fftA* with the MUI widgets discussed in Chapter **??**, we can write a simple program that generates a sine wave whose frequency is controlledd by a slider, and whose real-time graph as well as its FFT are displayed. The program to do this is shown in Figure 20.5.

20.4 Further Pragmatics

Exercise 20.5 Modify the program in Figure 20.5 in the following ways:

1. Add a second slider, and use it to control the frequency of a second oscillator.

2. Let s_1 and s_2 be the names of the signals whose frequencies are controlled by the first and second sliders, respectively. Instead of displaying the FFT of just s_1, try a variety of combinations of s_1 and s_2, such as $s_1 + s_2$, $s_1 - s_2$, $s_1 * s_2$, $1/s_1 + 1/s_2$, and s_1/s_2. Comment on the results.

3. Use s_2 to control the frequency of s_1 (as was done with *vibrato* in Chapter 19). Plot the fft of s_1 and comment on the result.

$$
\begin{array}{l}
\mathit{fftEx} :: \mathit{UISF}\ ()\ () \\
\mathit{fftEx} = \mathbf{proc}\ \_ \to \mathbf{do} \\
\quad f \leftarrow \mathit{hSlider}\ (1, 2000)\ 440 \prec ()\\
\quad (d, \_) \leftarrow \mathit{clockedSFToUISF}\ 100\ \mathit{simpleSig} \prec f \\
\quad \mathbf{let}\ (s, \mathit{fft}) = \mathit{unzip}\ d \\
\quad \_ \leftarrow \mathit{histogram}\ (500, 150)\ 20 \prec \mathit{listToMaybe}\ (\mathit{catMaybes}\ \mathit{fft}) \\
\quad \_ \leftarrow \mathit{realtimeGraph}'\ (500, 150)\ 200\ 20\ \mathit{Black} \prec s \\
\quad \mathit{outA} \prec () \\
\ \ \mathbf{where} \\
\quad \mathit{simpleSig} :: \mathit{SigFun}\ \mathit{CtrRate}\ \mathit{Double}\ (\mathit{Double}, \mathit{Event}\ [\mathit{Double}]) \\
\quad \mathit{simpleSig} = \mathbf{proc}\ f \to \mathbf{do} \\
\qquad s \leftarrow \mathit{osc}\ (\mathit{tableSinesN}\ 4096\ [1])\ 0 \prec f \\
\qquad \mathit{fftData} \leftarrow \mathit{fftA}\ 100\ 256 \prec s \\
\qquad \mathit{outA} \prec (s, \mathit{fftData}) \\
t_0 = \mathit{runMUI}\ (500, 600)\ \texttt{"fft Test"}\ \mathit{fftEx}
\end{array}
$$

Figure 20.5: A Real-Time Display of FFT Results

4. Instead of using *osc* to generate a pure sine wave, try using other oscillators and/or table generators to create more complex tones, and plot their FFT's. Comment on the results.

20.5 References

Most of the ideas in this chapter can be found in any good textbook on signal processing. The particular arrangement of the material here, in particular Figure 20.1 and the development and demonstration of a program for the DFT, is borrowed from the excellent text *Elements of Computer Music* by Moore [Moo90].

Chapter 21

Additive and Subtractive Synthesis

```
{-# LANGUAGE Arrows #-}
module Euterpea.Examples.Additive where
import Euterpea
```

There are many techniques for synthesizing sound. In this chapter we will discuss two of them: *additive synthesis* and *subtractive synthesis*. In practice it is rare for either of these, or any of the ones discussed in future chapters, to be utilized alone—a typical application may in fact employ all of them. But it is helpful to *study* them in isolation, so that the sound designer has a suitably rich toolbox of techniques at his or her disposal.

Additive synthesis is, conceptually at least, the simplest of the many sound synthesis techniques. Simply put, the idea is to add signals (usually sine waves of differing amplitudes, frequencies and phases) together to form a sound of interest. It is based on Fourier's theorem as discussed in the previous chapter, and indeed is sometimes called *Fourier synthesis*.

Subtractive synthesis is the dual of additive synthesis. The basic ideas is to start with a signal rich in harmonoc content, and seletively "remove" signals to create a desired effect.

In understanding the difference between the two, it is helpful to consider the following analogy to art:

- Additive synthesis is like painting a picture—each stroke of the brush, each color, each shape, each texture, and so on, adds to the artist's conception of the final artistic artifact.

- In contract, subtractive synthesis is like creating a sculpture from stone—each stroke of the chisel takes away material that is unwanted, eventually revealing the artist's conception of what the artistic artifact should be.

Additive synthesis in the context of Euterpea will be discussed in Section 21.1, and substractive synthesis in Section 21.2.

21.1 Additive Synthesis

21.1.1 Preliminaries

When doing pure additive synthesis it is often convenient to work with a *list of signal sources* whose elements are eventually summed together to form a result. To facilitate this, we define a few auxiliary functions, as shown in Figure 21.1.

constSF s sf simply lifts the value *s* to the signal function level, and composes that with *sf*, thus yielding a signal source.

foldSF f b sfs is analogous to *foldr* for lists: it returns the signal source *constA b* if the list is empty, and otherwise uses *f* to combine the results, pointwise, from the right. In other words, if *sfs* has the form:

$[sf1, sf2, ..., sfn]$

then the result will be:

$$
\begin{array}{l}
\textbf{proc}\ () \rightarrow \textbf{do} \\
\quad s_1 \leftarrow sf1 \prec ()\\
\quad s_2 \leftarrow sf2 \prec ()\\
\quad ...\\
\quad sn \leftarrow sfn \prec ()\\
\quad outA \prec f\ s_1\ (f\ s_2\ (...(f\ sn\ b)))
\end{array}
$$

$$
\begin{array}{l}
\mathit{constSF} :: \mathit{Clock}\ c \Rightarrow a \rightarrow \mathit{SigFun}\ c\ a\ b \rightarrow \mathit{SigFun}\ c\ ()\ b \\
\mathit{constSF}\ s\ \mathit{sf} = \mathit{constA}\ s \ggg \mathit{sf} \\
\mathit{foldSF} :: \mathit{Clock}\ c \Rightarrow \\
\qquad\qquad (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [\mathit{SigFun}\ c\ ()\ a] \rightarrow \mathit{SigFun}\ c\ ()\ b \\
\mathit{foldSF}\ f\ b\ \mathit{sfs} = \\
\quad \mathit{foldr}\ g\ (\mathit{constA}\ b)\ \mathit{sfs}\ \mathbf{where} \\
\qquad g\ \mathit{sfa}\ \mathit{sfb} = \\
\qquad\quad \mathbf{proc}\ ()\rightarrow \mathbf{do} \\
\qquad\qquad s_1 \leftarrow \mathit{sfa} \prec ()\\
\qquad\qquad s_2 \leftarrow \mathit{sfb} \prec ()\\
\qquad\qquad \mathit{outA} \prec f\ s_1\ s_2
\end{array}
$$

Figure 21.1: Working With Lists of Signal Sources

Details: *constSF* and *foldSF* are actually predefined in Euterpea, but with slightly more general types:

$$
\begin{array}{l}
\mathit{constSF} :: \mathit{Arrow}\ a \Rightarrow b \rightarrow a\ b\ d \rightarrow a\ c\ d \\
\mathit{foldSF}\ \ :: \mathit{Arrow}\ a \Rightarrow (b \rightarrow c \rightarrow c) \rightarrow c \rightarrow [a\ ()\ b] \rightarrow a\ ()\ c
\end{array}
$$

The more specific types shown in Figure 21.1 reflect how we will use the functions in this chapter.

21.1.2 Overtone Synthesis

Perhaps the simplest form of additive synthesis is combining a sine wave with some of its overtones to create a rich sound that is closer in harmonic content to that of a real instrument, as discussed in Chapter 18. Indeed, in Chapter 19 we saw several ways to do this using built-in Euterpea signal functions. For example, recall the function:

$$
\begin{array}{l}
\mathit{oscPartials} :: \mathit{Clock}\ c \Rightarrow \\
\qquad\qquad \mathit{Table} \rightarrow \mathit{Double} \rightarrow \mathit{SigFun}\ c\ (\mathit{Double}, \mathit{Int})\ \mathit{Double}
\end{array}
$$

oscPartials tab ph is a signal function whose pair of dynamic inputs determines the frequency, as well as the number of harmonics of that frequency, of the output. So this is a "built-in" notion of additive synthesis. A problem with this approach in modelling a conventional instrument is that the partials all have the same strength, which does not reflect the harmonic content of most physical instruments.

A more sophisticated approach, also described in Chapter 19, is based on various ways to build look-up tables. In particular, this function was defined:

$$\begin{array}{l} \mathit{tableSines3} :: \\ \quad \mathit{TableSize} \rightarrow [(\mathit{PartialNum}, \mathit{PartialStrength}, \mathit{PhaseOffset})] \rightarrow \mathit{Table} \end{array}$$

Recall that *tableSines3 size triples* is a table of size *size* that represents a sinusoidal wave and an arbitrary number of partials, whose relationship to the fundamental frequency, amplitude, and phase are determined by each of the triples in *triples*.

21.1.3 Resonance and Standing Waves

As we know from Fourier's Theorem, any periodic signal can be represented as a sum of a fundemental frequency and multiples of that fundamental frequency. We also know that a musical instrument's sound consists primarily of the sum of a fundamental frequency (the preceived pitch) and some of the multiples of that pitch (called harmonics, partials, or overtones). But what is it that makes a musical instrument behave this way in the first place? Answering this question can help us in understanding how to use additive synthesis to generate an instrument sound, but becomes even more important in Chapter **??** where we attempt to model the physical attributes of a particular instrument.

String Instruments

To answer this question, let's start with a simple string, fixed at both ends. Now imagine that energy is inserted at some point along the string—perhaps by a finger pluck, a guitar pick, a violin bow, or the hammer on a piano. This energy will cause the string to vibrate in some way. The energy will flow along the string as a wave, just like a pebble dropped in water, except that the energy only flows in one dimension, i.e. only along the orientation of the string. How fast the wave travels will depend on the string material and how taut it is. For example, the tauter the string, the faster the wave travels.

Because the ends of the string are fixed, however, the string can only vibrate in certain ways, which are called *modes*, or *resonances*. The most obvious mode for a string is shown in Figure 21.2a, where the center of the string is moving up and down, say, and the end-points do not move at all. Energy that is not directly contributing to a particular mode is quickly absorbed by the fixed endpoints. A mode is sometimes called a "standing wave" since it appears to be standing still—it does not seem to be moving up or down the string. But another way to think of it is that the energy in the string is being *reflected back* at each endpoint of the string, and those reflections reinforce each other to form the standing wave.

Eventually, of course, even the energy in a mode will dissipate, for three reasons: (1) since the ends of the string are never perfectly fixed, the reflections are not perfect either, and thus some energy is absorbed, (2) the movement of the string creates friction in the string material, generating heat and also absorbing energy, and (3) the transverse vibration of the string induces a longitudinal vibration in the air—i.e. the sound we hear—and that also absorbs some energy.

To better understand the nature of modes, suppose a pulse of energy is introduced at one end of the string. If v is the velocity of the resulting wave traveling along the string, and λ is the string length, then it takes λ/v seconds for a wave to travel the length of the string, and $p = 2\lambda/v$ for it to travel up and back. So if the pulse is repeated every p seconds, it will reinforce the previous pulse. If we think of p as the period of a periodic signal, its frequency in Hertz is the *reciprocol* of the period p, namely:

$$f_0 = v/(2\lambda)$$

Indeed, this is the frequency of the mode shown in Figure 21.2a, and corresponds to the fundamental frequency, i.e. the observed pitch.

Image not in repository!

Figure 21.2: The Modes of a Stringed Instrument

But note that this is not the only possible mode—another is shown in Figure 21.2b. This mode can be interpreted as repeating the pulse of energy inserted at the end of the string every $p/2$ seconds, thus corresponding to a frequency of:

$$f_1 = 1/(p/2) = v/\lambda = 2f_0$$

In other words, this is the first overtone.

Indeed, each subsequent mode corresponds to an overtone, and can be derived in the same way. A pulse of energy every p/n seconds corresponds to the (n-1)th overtone with frequency nf_0 Hz. Figure 21.2 shows these derivations for the first four modes; i.e. the fundamental plus three overtones.

Note: The higher overtones generally—but not always—decay more quickly primarily because they are generated by a quicker bending of the string, causing more friction and a quicker loss of energy.

Wind Instruments

Resonances in other musical instruments behave similarly. But in the case of a wind instrument, there a couple of important differences. First of all, the resonance happens within the air itelf, rather than a string. For example, a clarinet can be thought of as a *cylindical tube* closed at one end. The closed end is the mouthpiece, and the open end is called the "bell." The closed end, like the fixed end of a string, reflects energy directly back in the opposite direction. But because the open end is open, it behaves differently. In particular, as energy (a wave) escapes the open end, its pressure is dissipated into the air. This causes a pressure drop that induces a negative pressure—i.e. a vacumm—in the opposite direction, causing the wave to reflect back, *but inverted*!

Unfortuntely, we cannot easily visualize the standing wave in a clarinet, partly because the air is invisible, but also because, (1) the wave is *longitudinal*, whereas for a string it is transverse, and (2) as just discussed, the open end inverts the signal upon reflection. The best we can do is create a transverse representation. For example, Figure 21.3a represents the fundamental mode, or fundamantal frequency. Note that the left, closed end looks the same as for a fixed string—i.e. it is at the zero crossing of the sine wave. But the right end is different—it is intended to depict the inversion at the open end of the clarinet as the maximum absolute value of the sine wave. If the signal comes in at +1, it is inverted to the value -1, and so on.

Analogously to our detailed analysis of a string, we can analyze a clarinet's acoustic behavior as follows: Suppose a pulse of energy is introduced at the mouthpiece (i.e. closed end). If v is the velocity of sound in the air, and λ is the length of the clarinet, that wave appears at the open end in λ/v seconds. Its *inverted* reflection then appears back at the mouthpiece in $2 * \lambda/v$ seconds. But because it is inverted, *it will cancel out another pulse emitted* $2 * \lambda/v$ *seconds after the first!* On the other hand, suppose we let that reflection bounce off the closed end, travel back to the open end to be inverted a second time, and then return to the closed end. Two inversions are like no inversion at all, and so if we were to insert another pulse of energy at that moment, the two signals will be "in synch." In other words, if we repeat the pulse every $4\lambda/v$ seconds, the peaks and the troughs of the signals line up, and they will reinforce one another. This corresponds to a frequency of:

$$f_0 = v/(4\lambda)$$

and is in fact the fundamental mode, i.e. fundamental frequency, of the clarinet. This situation corresponds precisely to Figure 21.3a.

Image not in repository!

Figure 21.3: The Modes of a Clarinet Seen as a Cylindrical Tube

Now here is the interesting part: If we were to double the pulse rate in hopes of generating the first overtone, we arrive precisely at the situation we were in above: the signals cancel out. Thus, *a clarinet has no first overtone!* On the other hand, if we triple the pulse rate, the signals line up again, corresponding to a frequency of:

$$f_1 = v/((4/3)\lambda) = (3v)/(4\lambda) = 3f_0$$

This is the clarinet's second mode, and corresponds to Figure 21.3b.

By a similar argument, it can be shown that all the even overtones of a clarinet don't exist (or, equivalently, have zero amplitude), whereas all of the odd overtones do exist. Figure 21.3 shows the first three modes of a clarinet, corresponding to the fundamental frequency, and third and fifth overtones. (Note, by the way, the similarity of this to the spectral content of a square wave.)

[Todo: discuss other wind instruments]

Exercise 21.1 If $\omega = 2\pi f$ is the fundamental radial frequency, the sound of a sustained note for a typical clarinet can be approximated [] by:

$$\begin{aligned} s(t) \;\; = \;\; & \sin(\omega t) \; + \; 0.75\sin(3\omega t) \; + \; 0.5\sin(5\omega t) + 0.14\sin(7\omega t) \\ & + \; 0.5\sin(9\omega t) \; + \; 0.12\sin(11\omega t) \; + \; 0.17\sin(13\omega t) \end{aligned}$$

Define an instrument $clarinet :: Instr\ (Mono\ AudRate)$ that simulates this sound. Add an envelope to it to make it more realistic. Then test it with a simple melody.

21.1.4 Deviating from Pure Overtones

Sometimes, however, these built-in functions don't achieve exactly what we want. In that case, we can define our own, customized notion of additive synthesis, in whatever way we desire. For a simple example, traditional harmony is the simultaneous playing of more than one note at a time, and thus an instance of additive synthesis. More interestingly, richer sounds can be created by using slightly "out-of-tune" overtones; that is, overtones that are not an exact multiple of the fundamental frequency. For example:

```
-- TBD
```

This creates a kind of "chorusing" effect, very "electronic" in nature.

Some real instruments in fact exhibit this kind of behavior, and sometimes the degree of being "out of tune" is not quite fixed. Here's a variation of the above example where the detuning varies sinusoidally:

```
-- TBD
```

21.1.5 A Bell Sound

Synthesizing a bell or gong sound is a good example of "brute force" additive synthesis. Physically, a bell or gong can be thought of as a bunch of concentric rings, each having a different resonant frequency because they differ in diameter depending on the shape of the bell. Some of the rings will be more dominant than others, but the important thing to note is that these resonant frequencies often do not have an integral relationship with each other, and sometimes the higher frequencies can be quite strong, rather than rolling off significantly as with many other instruments. Indeed, it is sometime difficult to say exactly what the pitch of a particular bell is (especially large bells), so complex is its sound. Of course, the pitch of a bell can be controlled by mimimizing the taper of its shape (especially for small bells), thus giving it more of a pitched sound.

In any case, a pitched instrument representing a bell sound can be designed using additive synthesis by using the instrument's absolute pitch to create a series of partials that are conspicuously non-integral multiples of the fundamental. If this sound is then shaped by an envelope having a sharp rise time and a relatively slow, exponentially decreasing decay, we get a decent result. A Euterpea program to achieve this is shown in Figure 21.4. Note the use of *map* to create the list of partials, and *foldSF* to add them together. Also note that some of the partials are expressed as *fractions* of the fundamental—i.e. their frequencies are less than that of the fundamental!

The reader might wonder why we don't just use one of Euterpea's table generating functions, such as *tableSines3* discussed above, to generate a table with all the desired partials. The problem is, even though the

$$
\begin{array}{l}
bell_1 \;\; :: Instr\ (Mono\ AudRate) \\
\qquad\quad \texttt{--}\ Dur \to AbsPitch \to Volume \to AudSF\ ()\ Double \\
bell_1\ dur\ ap\ vol\ [\,] = \\
\quad \mathbf{let}\ f \;\;\; = apToHz\ ap \\
\qquad\quad v \;\;\; = fromIntegral\ vol / 100 \\
\qquad\quad d \;\;\; = fromRational\ dur \\
\qquad\quad sfs = map\ (\lambda p \to constA\ (f * p) \ggg osc\ tab_1\ 0) \\
\qquad\qquad\qquad\qquad [4.07, 3.76, 3, 2.74, 2, 1.71, 1.19, 0.92, 0.56] \\
\quad \mathbf{in\ proc}\ () \to \mathbf{do} \\
\qquad\quad aenv \leftarrow envExponSeg\ [0, 1, 0.001]\ [0.003, d - 0.003] \multimap\!\!\prec () \\
\qquad\quad a_1 \quad \leftarrow foldSF\ (+)\ 0\ sfs \multimap\!\!\prec () \\
\qquad\quad outA \multimap\!\!\prec a_1 * aenv * v / 9 \\
tab_1 = tableSinesN\ 4096\ [1] \\
bellTest1 = outFile\ \texttt{"bell1.wav"}\ 6\ (bell_1\ 6\ (absPitch\ (C, 5))\ 100\ [\,])
\end{array}
$$

Figure 21.4: A Bell Instrument

PartialNum argument to *tableSines3* is a *Double*, the normal intent is that the partial numbers all be integral. To see why, suppose 1.5 were one of the partial numbers—then 1.5 cycles of a sine wave would be written into the table. But the whole point of wavetable lookup synthesis is to repeatedly cycle through the table, which means that this 1.5 cycle would get repeated, since the wavetable is a periodic representation of the desired sound. The situation gets worse with partials such as 4.07, 3.75, 2.74, 0.56, and so on.

In any case, we can do even better than $bell_1$. An important aspect of a bell sound that is not captured by the program in Figure 21.4 is that the higher-frequency partials tend to decay more quickly than the lower ones. We can remedy this by giving each partial its own envelope (recall Section 19.3.2), and making the duration of the envelope inversely proportional to the partial number. Such a more sophisticated instrument is shown in Figure 21.5. This results in a much more pleasing and realistic sound.

Exercise 21.2 A problem with the more sophisticated bell sound in Figure 21.5 is that the duration of the resulting sound exceeds the specified duration of the note, because some of the partial numbers are less than one. Fix this.

Exercise 21.3 Neither of the bell sounds shown in Figures 21.4 and 21.5 actually contain the fundamental frequency—i.e. a partial number of 1.0.

$$
\begin{array}{l}
bell_2 \; :: Instr\ (Mono\ AudRate) \\
\qquad \texttt{--}\ Dur \rightarrow AbsPitch \rightarrow Volume \rightarrow AudSF\ ()\ Double \\
bell_2\ dur\ ap\ vol\ [\,] = \\
\quad \mathbf{let}\ f \;\; = apToHz\ ap \\
\qquad v \;\; = fromIntegral\ vol / 100 \\
\qquad d \;\; = fromRational\ dur \\
\qquad sfs = map\ (mySF\ f\ d) \\
\qquad\qquad\qquad [4.07, 3.76, 3, 2.74, 2, 1.71, 1.19, 0.92, 0.56] \\
\quad \mathbf{in\ proc}\ () \rightarrow \mathbf{do} \\
\qquad a_1 \quad \leftarrow foldSF\ (+)\ 0\ sfs \prec () \\
\qquad outA \prec a_1 * v / 9 \\
mySF\ f\ d\ p = \mathbf{proc}\ () \rightarrow \mathbf{do} \\
\quad s \qquad \leftarrow osc\ tab_1\ 0 \lll constA\ (f * p) \prec () \\
\quad aenv \leftarrow envExponSeg\ [0, 1, 0.001]\ [0.003, d / p - 0.003] \prec () \\
\quad outA \prec s * aenv \\
bellTest2 = outFile\ \texttt{"bell2.wav"}\ 6\ (bell_2\ 6\ (absPitch\ (C, 5))\ 100\ [\,])
\end{array}
$$

Figure 21.5: A More Sophisticated Bell Instrument

Yet they contain the partials at the integer multiples 2 and 3. How does this affect the result? What happens if you add in the fundamental?

Exercise 21.4 Use the idea of the "more sophisticated bell" to synthesize sounds other than a bell. In particular, try using only integral multiples of the fundamental frequency.

21.2 Subtractive Synthesis

As mentioned in the introduction to this chapter, subtractive synthesis involves starting with a harmonically rich sound source, and selectively taking away sounds to create a desired effect. In signal processing terms, we "take away" sounds using *filters*.

21.2.1 Filters

Filters can be arbitrarily complex, but are characterized by a *transfer function* that captures, in the frequency domain, how much of each frequency component of the input is transferred to the output. Figure 21.6 shows the general transfer function for the four most common forms of filters:

Image not in repository!

Figure 21.6: Transfer Functions for Four Common Filter Types

1. A *low-pass* filter passes low frequencies and rejects (i.e. attenuates) high frequencies.
2. A *high-pass* filter passes high frequencies and rejects (i.e. attenuates) low frequencies.
3. A *band-pass* filter passes a particular band of frequencies while rejecting others.
4. A *band-reject* (or *band-stop*, or *notch*) filter rejects a particular band of frequencies, while passing others.

It should be clear that filters can be combined in sequence or in parallel to achieve more complex transfer functions. For example, a low-pass and a high-pass filter can be combined in sequence to create a band-pass filter, and can be combined in parallel to create a band-reject filter.

In the case of a low-pass or high-pass filter, the *cut-off frequency* is usually defined as the point at which the signal is attenuated by 6dB. A similar strategy is used to define the upper and lower bounds of the band that is passed by a band-pass filter or rejected by a band-reject filter, except that the band is usually specified using a *center frequency* (the midpoint of the band) and a *bandwidth* the width of the band).

It is important to realize that not all filters of a particular type are alike. Two low-pass filters, for example, may, of course, have different cutoff frequencies, but even if the cutoff frequencies are the same, the "steepness" of the cutoff curves may be different (a filter with an ideal step curve for its transfer function does not exist), and the other parts of the curve might not be the same—they are never completely flat or even linear, and might not even be monotonically increasing or decreasing. (Although the diagrams in Figure 21.6 at least do not show a step curve, they are stll over-simplified in the smoothness and flatness of the curves.) Furthermore, all filters have some degree of *phase distortion*, which is to say that the transferred phase angle can vary with frequency.

$$\begin{array}{l}
\mathit{filterLowPass}, \mathit{filterHighPass}, \mathit{filterLowPassBW}, \mathit{filterHighPassBW} :: \\
\quad \mathit{Clock}\ p \Rightarrow \mathit{SigFun}\ p\ (\mathit{Double}, \mathit{Double})\ \mathit{Double} \\
\mathit{filterBandPass}, \mathit{filterBandStop} :: \\
\quad \mathit{Clock}\ p \Rightarrow \mathit{Int} \rightarrow \mathit{SigFun}\ p\ (\mathit{Double}, \mathit{Double}, \mathit{Double})\ \mathit{Double} \\
\mathit{filterBandPassBW}, \mathit{filterBandStopBW} :: \\
\quad \mathit{Clock}\ p \Rightarrow \mathit{SigFun}\ p\ (\mathit{Double}, \mathit{Double}, \mathit{Double})\ \mathit{Double}
\end{array}$$

Figure 21.7: Euterpea's Filters

In the digital domain, filters are often described using *recurrence equations* of varying degrees, and there is an elegant theory of filter design that can help predict and therefore control the various characteristics mentioned above. However, this theory is beyond the scope of this textbook. A good book on digital signal processing will elaborate on these issues in detail.

21.2.2 Euterpea's Filters

Instead of designing our own filters, we will use a set of pre-defined filters in Euterpea that are adequate for most sound synthesis applications. Their type sinatures are shown in Figure 21.7. As you can see, each of the filter types discussed previously is included, but their use requires a bit more explanation.

First of all, all of the filters ending in "*BW*" are what are called *Butterworth filters*, which are based on a second-order filter design that represents a good balance of filter characteristics: a good cutoff steepness, little phase distortion, and a reasonably flat response in both the pass and reject regions. Those filters without the *BW* suffix are first-order filters whose characteristics are not quite as good as the Butterworth filters, but are computationally more efficient.

In addition, the following points help explain the details of specific Euterpea filters:

- *filterLowPass* is a signal function whose input is a pair consisting of the signal being filtered, and the cutoff frequency (in that order). Note that this means the cutoff frequency can be varied dynamically. *filterHighPass*, *filterLowPassBW*, and *filterHighPassBW* behave analogously.
- *filterBandPassBW* is a signal function taking a triple as input: the

signal being filtered, the center frequency of the band, and the width of the band, in that order. For example:

$$\begin{array}{l} ... \\ \mathit{filterBandPassBW} \multimap (s, 2000, 100) \\ ... \end{array}$$

will pass the frequencies in s that are in the range 1950 to 2050 Hz, and reject the others. *filterBandStop* behaves analogously.

- *filterBandPass* and *filterBandStop* also behave analogously, except that they take a static *Int* argument, let's call it m, that has the following effect on the magnitude of the output:
  - $m = 0$ signifies no scaling of the output signal.
  - $m = 1$ signifies a peak response factor of 1; i.e. all frequencies other than the center frequency are attenuated in accordance with a normalized response curve.
  - $m = 2$ raises the response factor so that the output signal's overall RMS value equals 1.

21.2.3 Noisy Sources

Returning to the art metaphor at the beginning of this chapter, filters are like the chisels and other tools that a sculptor might use to fashion his or her work. But what about the block of stone that the sculptor begins with? What is the sound synthesis analogy to that?

The answer is some kind of a "noisy signal." It does not have to be pure noise in a signal processing sense, but in general its frequency spectrum will be rather broad and dense. Indeed, we have already seen (but not discussed) one way to do this in Euterpea: Recall the table generators *tableSines*, *tableSinesN*, *tableSines3*, and *tableSines3N*. When used with *osc*, these can generate very dense series of partials, which in the limit sound like pure noise.

In addition, Euterpea provides three sources of pure noise, that is, noise derived from a random number generator: *noiseWhite*, *noiseBLI*, and *noiseBLH*. More specifically:

1. $\mathit{noiseWhite} :: \mathit{Clock}\ p \Rightarrow \mathit{Int} \rightarrow \mathit{SigFun}\ p\ ()\ \mathit{Double}$
   noiseWhite n is a signal source that generates uniform white noise

with an RMS value of $1/\sqrt{2}$, where n is the "seed" of the underlying random number generator.

2. $noiseBLI :: Clock\ p \Rightarrow Int \rightarrow SigFun\ p\ Double\ Double$
   $noiseBLI\ n$ is like $noiseWhite\ n$ except that the signal samples are generated at a rate controlled by the (dynamic) input signal (presumably less than 44.1kHz), with interpolation performed between samples. Such a signal is called "band-limited" because the slower rate prevents spectral content higher than half the rate.

3. $noiseBLH :: Clock\ p \Rightarrow Int \rightarrow SigFun\ p\ Double\ Double$
   noiseBLH is like *noiseBLI* but does not interpolate between samples; rather, it "holds" the value fo the last sample.

21.2.4 Examples

```
sineTable :: Table
sineTable = tableSinesN 4096 [1]

env1 :: AudSF () Double
env1 = envExpon 20 10 10000
```

envExpon is better than *envLine* for sweeping a range of frequencies, because our ears hear pitches logarithmically. To demonstrate:

```
good = outFile "good.wav" 10
         (osc sineTable 0 <<< envExpon 20 10 10000 :: AudSF () Double)
bad = outFile "bad.wav" 10
         (osc sineTable 0 <<< envLine 20 10 10000 :: AudSF () Double)
```

Helper function for filter tests:

```
sfTest1 :: AudSF (Double, Double) Double -> Instr (Mono AudRate)
  -- AudSF (Double, Double) Double ->
  -- Dur -> AbsPitch -> Volume -> [Double] -> AudSF () Double
sfTest1 sf dur ap vol [] =
  let f = apToHz ap
      v = fromIntegral vol / 100
  in proc () -> do
       a1 <- osc sineTable 0 <<< env1 -< ()
       a2 <- sf -< (a1, f)
       outA -< a2 * v
```

Tests for low and highpass filters:

```
  tLow    = outFile "low.wav" 10 $
              sfTest1 filterLowPass 10 (absPitch (C, 5)) 80 []
  tHi     = outFile "hi.wav" 10 $
              sfTest1 filterHighPass 10 (absPitch (C, 5)) 80 []
  tLowBW  = outFile "lowBW.wav" 10 $
              sfTest1 filterLowPassBW 10 (absPitch (C, 5)) 80 []
  tHiBW   = outFile "hiBW.wav" 10 $
              sfTest1 filterHighPassBW 10 (absPitch (C, 5)) 80 []
```

Tests for bandpass and bandstop filters (varying center frequency):

```
addBandWidth :: AudSF (Double, Double, Double) Double →
                AudSF (Double, Double) Double
addBandWidth filter =
  proc (a, f) → do filter ⤙ (a, f, 200)
tBP    = outFile "bp.wav" 10 $
           sfTest1 (addBandWidth (filterBandPass 1)) 10 (absPitch (C, 6)) 80 []
tBS    = outFile "bs.wav" 10 $
           sfTest1 (addBandWidth (filterBandStop 1)) 10 (absPitch (C, 6)) 80 []
tBPBW  = outFile "bpBW.wav" 10 $
           sfTest1 (addBandWidth filterBandPassBW) 10 (absPitch (C, 6)) 80 []
tBSBW  = outFile "bsBW.wav" 10 $
           sfTest1 (addBandWidth filterBandStopBW) 10 (absPitch (C, 6)) 80 []
```

Pure white noise:

```
 noise1 :: Instr (Mono AudRate)
       -- Dur → AbsPitch → Volume → [Double] → AudSF () Double
 noise1 dur ap vol [] =
   let v = fromIntegral vol/100
   in proc () → do
        a1  ← noiseWhite 42 ⤙ ()
        outA ⤙ a1 ∗ v
 test1 = outFile "noise1.wav" 6 (noise1 6 (absPitch (C, 5)) 100 [])
```

Tests for bandpass and bandstop filters (varying bandwidth):

```
env2 :: AudSF () Double
env2 = envExpon 1 10 2000
sfTest2 :: AudSF (Double, Double, Double) Double → Instr (Mono AudRate)
     -- AudSF (Double, Double, Double) Double →
```

```
    -- Dur → AbsPitch → Volume → [Double] → AudSF () Double
sfTest2 sf dur ap vol [] =
  let f = apToHz ap
      v = fromIntegral vol/100
  in proc () → do
      a1 ← noiseWhite 42 −≺ ()
      bw ← env2 −≺ ()
      a2 ← sf −≺ (a1, f, bw)
      outA −≺ a2

      tBP'    = outFile "bp'.wav" 10 $
                  sfTest2 (filterBandPass 1) 10 (absPitch (C, 5)) 80 []
      tBS'    = outFile "bs'.wav" 10 $
                  sfTest2 (filterBandStop 1) 10 (absPitch (C, 5)) 80 []
      tBPBW' = outFile "bpBW'.wav" 10 $
                  sfTest2 filterBandPassBW 10 (absPitch (C, 5)) 80 []
      tBSBW' = outFile "bsBW'.wav" 10 $
                  sfTest2 filterBandStopBW 10 (absPitch (C, 5)) 80 []
```

Bandlimited noise:

```
noise2 :: Instr (Mono AudRate)
noise2 dur ap vol [] =
  let f = apToHz ap
      v = fromIntegral vol/100
  in proc () → do
      a1    ← noiseBLI 42 −≺ f
      outA −≺ a1 ∗ v
test2 = outFile "noise2.wav" 6 (noise2 6 (absPitch (C, 5)) 100 [])
```

Simple subtractive synthesis:

```
ss1 :: Instr (Mono AudRate)
ss1 dur ap vol [] =
  let v = fromIntegral vol/100
  in proc () → do
      a1    ← noiseWhite 42 −≺ ()
      a2    ← filterBandPass 2 −≺ (a1, 1000, 200)
      outA −≺ a2 ∗ v/5
test3 = outFile "ss1.wav" 6 (ss1 6 (absPitch (C, 5)) 100 [])
```

Howling wind:

```
wind :: Instr (Mono AudRate)
wind dur ap vol [ ] =
  let f = apToHz ap
      v = fromIntegral vol / 100
  in proc () -> do
      a1   <- noiseWhite 42 -< ()
      lfo1 <- osc sineTable 0 -< 0.9
      lfo2 <- osc sineTable 0 -< 1.3
      a2   <- filterBandPass 2 -< (a1, f + 100 * (lfo1 + lfo2), 200)
      outA -< a2 * v / 5
test4 = outFile "wind.wav" 6 (wind 6 (absPitch (C, 7)) 100 [ ])
```

Dense partials ("buzz")

```
buzzy :: Instr (Mono AudRate)
buzzy dur ap vol [ ] =
  let f = apToHz ap
      v = fromIntegral vol / 100
  in proc () -> do
      a1 <- oscPartials sineTable 0 -< (f, 20)
      outA -< a1 * v
test5 = outFile "buzzy.wav" 6 (buzzy 6 (absPitch (C, 5)) 100 [ ])
```

Dense partials filtered and Shaped:

```
buzzy2 :: Instr (Mono AudRate)
buzzy2 dur ap vol [ ] =
  let f   = apToHz ap
      v   = fromIntegral vol / 100
      d   = fromRational dur
  in proc () -> do
      a1  <- oscPartials sineTable 0 -< (f, 20)
      env <- envExponSeg [0, 1, 0.001] [0.003, d - 0.003] -< ()
      a2  <- filterLowPass -< (a1, 20000 * env)
      outA -< a2 * v * env
test6 = outFile "buzzy2.wav" 6 (buzzy2 6 (absPitch (C, 5)) 100 [ ])
```

Sci-Fi-1:

```
scifi1 :: Instr (Mono AudRate)
scifi1 dur ap vol [ ] =
  let v = fromIntegral vol / 100
  in proc () -> do
```

$\quad\quad a_1 \leftarrow noiseBLH\ 42 \prec 8$
$\quad\quad a_2 \leftarrow osc\ sineTable\ 0 \prec 600 + 200 * a_1$
$\quad\quad outA \prec a_2 * v$
$test7 = outFile$ `"scifi1.wav"` $10\ (scifi1\ 10\ (absPitch\ (C, 5))\ 100\ [\,])$

Sci-Fi-2:

$scifi2 :: Instr\ (Mono\ AudRate)$
$scifi2\ dur\ ap\ vol\ [\,] =$
$\quad$ **let** $v = fromIntegral\ vol / 100$
$\quad$ **in proc** $() \rightarrow$ **do**
$\quad\quad a_1 \leftarrow noiseBLI\ 44 \prec 8$
$\quad\quad a_2 \leftarrow osc\ sineTable\ 0 \prec 600 + 200 * a_1$
$\quad\quad outA \prec a_2 * v$
$test8 = outFile$ `"scifi2.wav"` $10\ (scifi2\ 10\ (absPitch\ (C, 5))\ 100\ [\,])$

‘

Chapter 22

Amplitude and Frequency Modulation

To *modulate* something is to change it in some way. In signal processing, *amplitude modulation* is the process of modifying a signal's amplitude *by another signal*. Similarly, *frequency modulation* is the process of modifying a signal's frequency *by another signal*. These are both powerful sound synthesis techniques that will be discussed in this chapter.

22.1 Amplitude Modulation

Technically speaking, whenever the amplitude of a signal is dynamically changed, it is a form of *amplitude modulation*, or *AM* for short; that is, we are modulating the amplitude of a signal. So, for example, shaping a signal with an envelope, as well as adding tremolo, are both forms of AM. In this section more interesting forms of AM are explored, including their mathematical basis. To help distinguish these forms of AM from others, we define a few terms:

- The dynamically changing signal that is doing the modulation is called the *modulating signal*.
- The signal being modulated is sometimes called the *carrier*.
- A *unipolar signal* is one that is always either positive or negative (usually positive).

- A *bipolar signal* is one that takes on both positive and negative values (that are often symmetric and thus average out to zero over time).

So, shaping a signal using an envelope is an example of amplitude modulation using a unipolar modulating signal whose frequency is very low (to be precise, $1/dur$, where *dur* is the length of the note), and in fact only one cycle of that signal is used. Likewise, tremolo is an example of amplitude modulation with a unipolar modulating signal whose frequency is a bit higher than with envelope shaping, but still quite low (typically 2-10 Hz). In both cases, the modulating signal is infrasonic.

Note that a bipolar signal can be made unipolar (or the other way around) by adding or subtracting an offset (sometimes called a "DC offset," where DC is shorthand for "direct current"). This is readily seen if we try to mathematically formalize the notion of tremolo. Specifically, tremolo can be defined as adding an offset of 1 to an infrasonic sine wave whose frequency is f_t (typically 2-10Hz), multiplying that by a "depth" argument d (in the range 0 to 1), and using the result as the modulating signal; the carrier frequency is f:

$$(1 + d \times \sin(2\pi f_t t)) \times \sin(2\pi f t)$$

Based on this equation, here is a simple tremolo envelope generator written in Euterpea, and defined as a signal source (see Exercise 19.2):

```
tremolo :: Clock c ⇒
           Double → Double → SigFun c () Double
tremolo tfrq dep = proc () → do
  trem ← osc tab₁ 0 −≺ tfrq
  outA −≺ 1 + dep ∗ trem
```

tremolo can then be used to modulate an audible signal as follows:

```
-- TBD
```

22.1.1 AM Sound Synthesis

But what happens when the modulating signal is audible, just like the carrier signal? This is where things get interesting from a sound synthesis point of view, and can result in a rich blend of sounds. To understand this mathematically, recall this trigonometric identity:

$$\sin(C) \times \sin(M) = \frac{1}{2}(\cos(C - M) - \cos(C + M))$$

or, sticking entirely with cosines:

$$\cos(C) \times \cos(M) = \frac{1}{2}(\cos(C - M) + \cos(C + M))$$

These equations demonstrate that AM in a sense is just a form additive synthesis. Indeed, the equations imply two ways to implement AM in Euterpea: We could directly multiply the two outputs, as specified by the left-hand sides of the equations above, or we could add two signals as specified by the right-hand sides of the equations.

Note the following:

1. When the modulating frequency is the same as the carrier frequency, the right-hand sides above reduce to $1/2 \cos(2C)$. That is, we essentially double the frequency.

2. Since multiplication is commutative, the following is also true:

   $$\cos(C) \times \cos(M) = \frac{1}{2}(\cos(M - C) + \cos(M + C))$$

   which is valid because $\cos(t) = \cos(-t)$.

3. Scaling the modulating signal or carrier just scales the entire signal, since multiplication is associative.

Also note that adding a third modulating frequency yields the following:

$$\begin{aligned}
&\cos(C) \times \cos(M1) \times cos(M2) \\
&\quad = (0.5 \times (\cos(C - M1) \times \cos(C + M1))) \times \cos(M2) \\
&\quad = 0.5 \times (\cos(C - M1) \times \cos(M2) + \cos(C + M1) \times \cos(M2)) \\
&\quad = 0.25 \times (\cos(C - M1 - M2) + \cos(C - M1 + M2) + \\
&\qquad\quad \cos(C + M1 - M2) + \cos(C + M1 + M2))
\end{aligned}$$

In general, combining n signals using amplitude modulation results in 2^{n-1} signals. AM used in this way for sound synthesis is sometimes called *ring modulation*, because the analog circuit (of diodes) originally used to implement this technique took the shape of a ring. Some nice "bell-like" tones can be generated with this technique.

22.1.2 What do Tremolo and AM Radio Have in Common?

Combining the previous two ideas, we can use a bipolar carrier in the *electromagnetic spectrum* (i.e. the radio spectrum) and a unipolar modulating frequency in the *audible* range, which we can represent mathematically as:

$$\cos(C) \times (1 + \cos(M)) = \cos(C) + 0.5 \times (\cos(C - M) + \cos(C + M))$$

Indeed, this is how AM radio works. The above equation says that AM radio results in a carrier signal plus two sidebands. To completely cover the audible frequency range, the modulating frequency would need to be as much as 20kHz, thus yielding sidebands of $\pm$20kHz, thus requiring station separation of at least 40 kHz. Yet, note that AM radio stations are separated by only 10kHz! (540 kHz, 550 kHz, ..., 1600 kHz). This is because, at the time commercial AM radio was developed, a fidelity of 5KHz was considered "good enough."

Also note now that the amplitude of the modulating frequency does matter:

$$\cos(C) \times (1 + A \times cos(M)) = cos(C) + 0.5 \times A \times (\cos(C - M) + \cos(C + M))$$

A, called the *modulation index*, controls the size of the sidebands. Note the similarity of this equation to that for tremolo.

Exercise 22.1 Experiment with amplitude modulation as a sound synthesis technique. In particular, try using three or more signals.

22.2 Frequency Modulation

[This section needs work.]

As mentioned in the introduction of this chapter, if we modulate the frequency (rather than amplitude) of a signal with another signal, we are performing *frequency modulation*, or FM.

FM can be used to implement vibrato. [show code] Contrast this with tremolo, which varies the amplitude. [show code?] It is also the basis for FM radio. [see later subsection]

More generally, FM with two audible signals causes complex things happen, and results in a much richer sound than AM. In its simplest form, we can express FM as follows:

$$A\cos(C + d\cos(M))$$

where, as for AM, C is the carrier frequency, and M is the modulating frequency (both in radians).

The result of FM is a carrier plus a large number of sidebands that are integer multiples of the difference between C and M, as depicted in Figure 22.1. But how many and how large are these sidebands?

22.2.1 Bessell Functions

Let's rewrite the previous formula more formally as follows:

$$s_{fm}(t) = A\cos(2\pi(f_c + d\cos(2\pi f_m\ t))t)$$

where f_c and f_m are the frequencies, in Hertz, of the carrier and modulating signals, respectively. As we will see, A and d relate to the magnitude of the carrier and sidebands.

The *FM modulation index I* is defined as:

$$I = D/f_m$$

where D is the maximum deviation in Hertz of the carrier. For example, if the modulating signal f_m is a pure sine wave, then it varies between $+1$ and -1, so the maximum deviation would be d, and therefore $I = d/f_m$.

The equation for s_{fm} above is difficult to understand, being the cosine of a cosine, but it can be shown to be equivalent to the following sum of sines:

$$s_{fm}(t) = \sum_{n=-\infty}^{+\infty} J_n(I)\ \sin(2\pi(f_c \pm n f_m)\ t)$$

where J_n is the nth *Bessell function*. Ignoring the scaling effect of the Bessell function for a moment, note that the resulting signal consists of the carrier (at $n = 0$) plus sidebands spaced evenly at increments of the modulating frequency f_m.

Bessell functions themselves are beyond the scope of this text, but can be visualized as shown in Figure 22.2. In this figure, note that:

- ...

22.2.2 FM Sound Synthesis

From a sound synthesis point of view, it is important to realize that, for many carrier and modulating frequency pairs, the spectrum of the result is quite complex, far more than for, say, amplitude modulation or additive synthesis. Indeed, according to equation **??**, there are an infinite number of sidebands.

The generic Euterpea arrow-code snippet corresponding to Equation **??** is:

$$
\begin{array}{l}
\ldots \\
ms \leftarrow osc \ldots \prec mf \\
s \quad \leftarrow osc \ldots \prec cf + d * ms \\
outA \prec a * s \\
\ldots
\end{array}
$$

where *cf* and *mf* are the carrier and modulating frequencies, respectively, and *ms* is the modulating signal.

[show UISF example; explain d, etc.]

[give examples where the carrier, modulating frequency, and modulation index are "swept" appropriately]

Here are some "rules of thumb" that are useful in selecting the modulation index:

- To synthesize harmonic sounds, the modulating signal should have a harmonic relationship to the original carrier signal.
- Use of modulating frequencies that are non-integer multiples of the carrier results in inharmonic sounds, which are nevertheless interesting and pleasant sounding in their own way.
- A low modulation index results in sidebands that are "tight" near the carrier, but the variations between them can be large.
- A higher mod index results in more sidebands at a greater distance from the carrier, but they are more uniform in size.
- In addition, as I increases, the drop-off in sidebands is not uniform; in fact it is sinusoidal, so there are regions where the sidebands drop out completely, then reappear further away from the carrier.
- *Carson's rule* states that nearly all (98 percent) of the power of a

frequency-modulated signal lies within a bandwidth of:

$$B_T = 2(\Delta f + f_m)$$

where Δf, is the peak deviation of the instantaneous frequency from the carrier frequency f_c.

FM synthesis is a powerful methodolgy for musical sound design. There are many ways to create both real musical instrument sounds and ethereal ones. Even the seminal paper on FM sound synthesis [Cho73] written over 40 years ago gives great insight into this.

22.2.3 An FM Bell

By creating inharmonic sounds, atonal and tonal bell-like and percussive sounds can easily be created. Figure 22.3 shows a Euterpea program for an FM bell. Note that the pitch f of the bell is used as both the carrier frequency, and, slightly modified, as the modulating frquency. That modulating frequency varies sinusoidally from ... Note that d in this case is ... This is strange!

22.2.4 FM Radio

Reword:

The FM broadcasting range is 87.5-108 MHz. It uses a channel spacing of 200 kHz, with a maximum frequency deviation of 75 kHz, leaving a 25 kHz buffer above the highest and below the lowest frequency to reduce interaction with other channels.

Figure 22.1: FM Spectrum

Figure 22.2: Three-Dimensional Graph of First Fifteen Bessell Functions

.

```
bellFM :: Instr (Mono AudRate)
bellFM dur ap vol [] =
  let f = apToHz ap
      v = fromIntegral vol/100
      d = fromRational dur
  in proc () → do
      aenv ← envExponSeg [0, 1, 0.001] [0.003, d − 0.003] ⤙ ()
      a2 ← osc tab1 0 ⤙ 280
      a1 ← osc tab1 0 ⤙ f + (a2 ∗ f ∗ 0.95)
      outA ⤙ a1 ∗ aenv ∗ v
bellTest = outFile "bellFM.wav" 6 (bellFM 6 (absPitch (G, 4)) 80 [])
```

Figure 22.3: An FM Bell

Chapter 23

Physical Modelling

23.1 Introduction

...

23.2 Delay Lines

An important tool for physical modeling is the *delay line*. In this section we will discuss the basic concepts of a delay line, the delay line signal functions in Eutperpea, and a couple of fun examples that do not yet involve physical modeling.

Conceptually, a delay line is fairly simple: it delays a signal by a certain number of seconds (or, equivalently, by a certain number of samples at some given sample rate). Figure **??**(a) show this pictorially—if s is the number of samples in the delay line, and r is the clock rate, then the delay d is given by:

$$d = s/r$$

In the case of audio, of course, r will be 44.1 kHz. So to achieve a one-second delay, s would be chosen to be 44,100. In essence, a delay line is a *queue* or *FIFO* data structure.

In Euterpea there is a family of delay lines whose type signatures are given in Figure 23.1. Their behaviors can be described as follows:

- *init* x is a delay line with one element, which is initalized to x.

Unit delay line:

$$\begin{array}{l} init :: Clock\ c \Rightarrow \\ \quad Double \rightarrow SigFun\ c\ Double\ Double \end{array}$$

Fixed-length delay line, initialized with a table:

$$\begin{array}{l} delayLineT :: Clock\ c \Rightarrow \\ \quad Int \rightarrow Table \rightarrow SigFun\ c\ Double\ Double \end{array}$$

Fixed-length delay line, initialized with zeros:

$$\begin{array}{l} delayLine :: Clock\ c \Rightarrow \\ \quad Double \rightarrow SigFun\ c\ Double\ Double \end{array}$$

Delay line with variable tap:

$$\begin{array}{l} delayLine1 :: Clock\ c \Rightarrow \\ \quad Double \rightarrow SigFun\ c\ (Double, Double)\ Double \end{array}$$

Figure 23.1: Euterpea's Delay Lines

- *delayLineT s tab* is a delay line whose length is s and whose contents are initialized to the values in the table *tab* (presumably of length s).
- *delayLine d* is a delay line whose length achieves a delay of d seconds.
- *delayLine1 d* is a "tapped delay line" whose length achieves a maximum delay of d seconds, but whose actual output is a "tap" that results in a delay of somewhere between 0 and d seconds. The tap is controlled dynamically. For example, in:

  ...
  $out \leftarrow delayLine1\ d \prec (s, t)$
  ...

  s is the input signal, and t is the tap delay, which may vary between 0 and d.

Before using delay lines for physical modelling, we will explore a few simple application that should give the reader a good sense of how they work.

Image not in repository!

Figure 23.2: Delay Line Examples

23.2.1 Simulating an Oscillator

Let's contrast a delay line to the oscillators introduced in Chapter 19 that are initialized with a table (like *osc*). These oscillators cycle repetitively through a given table at a variable rate. Using *delayT*, a delay-line can also be initialized as a table, but it is processed at a fixed rate (i.e. the clock rate)—at each step, one value goes in, and one goes out.

Nevertheless, we can simulate an oscillator by initializing the delay line with one cycle of a sine wave, and "feeding back" the output to the input, as shown in Figure 23.2a. At the standard audio sample rate, if the table size is s, then it takes $s/44,100$ seconds to output one cycle, and therefore the resulting frequency is the reciprocol of that:

$$f = 44,100/s$$

There is one problem, however: when coding this idea in Haskell, we'd like to write something like:

```
...
x ← delayLineT s table ⤙ x
...
```

However, arrow syntax in its standard form does not allow recursion! Fortunately, arrow syntax supports a keyword **rec** that allows us to specify where recursion takes place. For example to generate a tone of 441 Hz we need a table size of 44,100/441 = 100, leading to:

```
sineTable441 :: Table
sineTable441 = tableSinesN 100 [1]

s441 :: AudSF () Double
s441 = proc () → do
  rec s ← delayLineT 100 sineTable441 ⤙ s
  outA ⤙ s

ts441 = outFile "s441.wav" 5 s441
```

Details: Say more about the **rec** keyword.

23.2.2 Echo Effect

Perhaps a more obvious use of a delay line is to simply delay a signal! But to make this more exciting, let's go one step further and *echo* the signal, using feedback. To prevent the signal from echoing forever, let's decay it a bit each time it is fed back. A diagram showing this strategy is shown in Figure 23.2b, and the resulting code is:

```
echo :: AudSF Double Double
echo = proc s → do
  rec fb ← delayLine 0.5 −≺ s + 0.7 ∗ fb
  outA −≺ fb / 3
```

Here the delay time is 0.5 seconds, and the decay rate is 0.7.

[test code?]

23.2.3 Modular Vibrato

Recall that we previously defined a tremolo signal function that could take an arbitrary signal and add tremolo. This is because tremolo simply modulates the amplitude of a signal, which could be anything—music, speech, whatever—and can be done after that sound is generated. So we could define a function:

```
tremolo :: Rate → Depth → AudSF Double Double
```

to achieve the result we want.

Can we do the same for vibrato? In the version of vibrato we defined previously (see Section **??**), we used frequency modulation—but that involved modulating the actual frequency of a specific oscillator, *not* the output of the oscillator that generated a sine wave of that frequency. So using that technique, at least, it doesn't seem possible to define a function such as:

```
vibrato :: Rate → Depth → AudSF Double Double
```

that would achieve our needs. Indeed, if we were using additive synthesis, one might imagine having to add vibrato to every sine wave that makes up the result. Not only is this a daunting task, but, in effect, we would lose modularity!

But in fact we can define a "modular" vibrato using a delay line with a variable tap. The idea is this: Send a signal into a tapped delay line, adjust the initial tap to the center of that delay line, and then move it back and forth sinusoidally at a certain rate to control the frequency of the vibrato, Image not in repository!

Figure 23.3: Karplus-Strong and Waveguides

and move it a certain distance (with a maximum of one-half the delay line's maximum delay) to achieve the depth. This idea is shown pictorially in Figure 23.2c, and the code is given below:

```
vibrato :: Rate → Depth → AudSF Double Double
modVib :: AudSF Double Double
modVib rate depth =
  proc sin → do
    vib  ← osc sineTable 0 −≺ rate
    sout ← delayLine1 0.2 −≺ (sin, 0.1 + 0.005 ∗ vib)
    outA −≺ sout
tModVib = outFile "modvib.wav" 6 $
  constA 440 ⋙ osc sineTable 0 ⋙ vibrato 5 0.005
```

–

[discuss problem with "noisy" output, and with initial delay]

23.3 Karplus-Strong Algorithm

Now that we know how delay lines work, let's look at their use in physical modeling. The *Karplus-Strong Algorithm* [KS83] was one of the first algorithms classified as "physical modeling." It's a good model for synthesizing plucked strings and drum-like sounds. The basic idea is to use a recursive delay line to feed back a signal onto itself, thus simulating the standing wave modes discussed in Section 21.1.3. The result is affected by the initial values in the delay line, the length of the delay line, and any processing in the feedback loop. A diagram that depicts this algorithm is shown in Figure 23.3(a).

23.3.1 Physical Model of a Flute

Figure 23.4 shows a physical model of a flute, based on the model of a "slide flute" proposed by Perry Cook in [Coo02]. Although described as a slide flute, it sounds remarkably similar a regular flute. Note that the lower right part of diagram looks just like the feedback loop in the Karplus-

Image not in repository!

Figure 23.4: A Physical Model of a Flute

Strong algorithm. The rest of the diagram is intended to model the breath, including vibrato, which drives a "pitched" embouchure that in turn drives the flute bore.

The Euterpea code for this model is essentially a direct translation of the diagram, with details of the envelopes added in, and is shown in Figure 23.5. Some useful test cases are:

$f_0 = \mathit{flute}$ 3 0.35 440 0.93 0.02 -- average breath
$f_1 = \mathit{flute}$ 3 0.35 440 0.83 0.05 -- weak breath, soft note
$f_2 = \mathit{flute}$ 3 0.35 440 0.53 0.04 -- very weak breath, no note

23.4 Waveguide Synthesis

The Karplus-Strong algorithm can be generalized to a more accurate model of the transmission of sound up and down a medium, whether it be a string, the air in a chamber, the surface of a drum, the metal plate of a xylophone, and so on. This more accurate model is called a *waveguide*, and, mathematically, can be seen as a discrete model of d'Alembert's solution to the *one-dimensional wave equation*, which captures the superposition of a right-going wave and a left-going wave, as we have discussed earlier in Section 21.1.3. In its simplest form, we can express the value of a wave at position m and time n as:

$$y(m, n) = y^{+}(m - n) + y^{-}(m + n)$$

where y^{+} is the right-going wave and y^{-} is the left-going wave. Intuitively, the value of y at point m and time n is the sum of two delayed copies of its traveling waves. As discusses before, these traveling waves will reflect at boundaries such as the fixed ends of a string or the open or closed ends of tubes.

What distinguishes this model from the simpler Karplus-Strong model is that it captures waves traveling in *both* directions—and to realize that, we need a closed loop of delay lines. But even with that generalization, the equation above assumes a *lossless* system, and does not account for interactions *between* the left- and right-traveling waves. The former quantity is

```haskell
flute :: Time -> Double -> Double -> Double -> Double
        -> AudSF () Double
flute dur amp fqc press breath =
  proc () -> do
    env1  <- envLineSeg [0, 1.1 * press, press, press, 0]
                        [0.06, 0.2, dur - 0.16, 0.02] -< ()
    env2  <- envLineSeg [0, 1, 1, 0]
                        [0.01, dur - 0.02, 0.01]      -< ()
    envib <- envLineSeg [0, 0, 1, 1]
                        [0.5, 0.5, dur - 1]           -< ()
    flow  <- noiseWhite 42  -< ()
    vib   <- osc sineTable 0 -< 5
    let emb = breath * flow * env1 + env1 + vib * 0.1 * envib
    rec flute <- delayLine (1/fqc)     -< out
        x     <- delayLine (1/fqc/2) -< emb + flute * 0.4
        out   <- filterLowPassBW -< (x - x * x * x + flute * 0.4, 2000)
    outA -< out * amp * env2

sineTable :: Table
sineTable = tableSinesN 4096 [1]
```

Figure 23.5: Euterpea Program for Flute Model

often called the *gain*, and the latter the *reflection coefficient*. We have discussed previously the notion that waves are reflected at the ends of a string, a tube, etc., but in general some interaction/reflection between the left- and right-traveling waves can happen anywere. This more general model is shown diagramatically in Figure 23.3(c), where g is the gain, and r is the reflection coefficient.

Figure 23.3(b) shows a sequence of waveguides "wired together" to allow for the possibility that the gain and reflection characteristics are different at different points along the medium. The "termination" boxes can be thought of as special waveguides that capture the effect of reflection at the end of a string, tube, etc.

23.4.1 Waveguides in Euterpea

A simple waveguide with looping delay lines and gain factors, but that ignores reflection, is shown below:

$$
\begin{aligned}
&\mathit{waveguide} :: \mathit{Double} \rightarrow \mathit{Double} \rightarrow \mathit{Double} \rightarrow \\
&\quad \mathit{AudSF}\ (\mathit{Double}, \mathit{Double})\ (\mathit{Double}, \mathit{Double}) \\
&\mathit{waveguide}\ \mathit{del}\ \mathit{ga}\ \mathit{gb} = \mathbf{proc}\ (\mathit{ain}, \mathit{bin}) \rightarrow \mathbf{do} \\
&\quad \mathbf{rec}\ \mathit{bout} \leftarrow \mathit{delayLine}\ \mathit{del} \prec \mathit{bin} - \mathit{ga} * \mathit{aout} \\
&\qquad \mathit{aout} \leftarrow \mathit{delayLine}\ \mathit{del} \prec \mathit{ain} - \mathit{gb} * \mathit{bout} \\
&\quad \mathit{outA} \prec (\mathit{aout}, \mathit{bout})
\end{aligned}
$$

Here ga and gb are the gains, and del is the delay time of one delay line.

This waveguide is good enough for the examples studied in this book, in that we assume no reflections occur along the waveguide, and that reflections at the end-points can be expressed manually with suitable feedback. Similarly, any filtering or output processing can be expressed manually.

23.4.2 A Pragmatic Issue

In a circuit with feedback, DC ("direct current") offsets can accumulate, resulting in clipping. The DC offset can be "blocked" using a special high-pass filter whose cutoff frequency is infrasonic. This filter can be captured by the difference equation:

$$y[n] = x[n] - x[n-1] + a * y[n-1]$$

Where $x[n]$ and $y[n]$ are the input and output, respectively, at the current

```
dcBlock :: Double → AudSF Double Double
dcBlock a = proc xn → do
  rec let yn = xn − xn_1 + a ∗ yn_1
        xn_1 ← init 0 −≺ xn
        yn_1 ← init 0 −≺ yn
  outA −≺ yn
```

Figure 23.6: DC Blocking Filter in Euterpea

Image not in repository!

Figure 23.7: Transfer Function for DC Blocker

time n, and a is called the gain factor. If we think of the indexing of n versus $n-1$ as a one unit delay, we can view the equation above as the diagram shown in Figure **??**, where z^{-1} is the mathematical notation for a 1-unit delay.[1]

If you recall from Section21.2.2, the one-unit delay operator in Euterpea is called *init*. With that in mind, it is easy to write a Euterpea program for a DC blocking filter, as shown in Figure 23.6. The transfer function corresponding to this filter for different values of a is shown in Figure 23.7. In practice, a value of 0.99 for a works well.

[1] This representation is called the *Z transform* of a signal.

Chapter 24

Sound Effects

TBD

Appendix

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:

$$
\begin{array}{l}
\textbf{module}\ \mathit{PreludeList}\ (\\
\quad \mathit{map}, (+\!\!+), \mathit{filter}, \mathit{concat}, \\
\quad \mathit{head}, \mathit{last}, \mathit{tail}, \mathit{init}, \mathit{null}, \mathit{length}, (!!), \\
\quad \mathit{foldl}, \mathit{foldl1}, \mathit{scanl}, \mathit{scanl1}, \mathit{foldr}, \mathit{foldr1}, \mathit{scanr}, \mathit{scanr1}, \\
\quad \mathit{iterate}, \mathit{repeat}, \mathit{replicate}, \mathit{cycle}, \\
\quad \mathit{take}, \mathit{drop}, \mathit{splitAt}, \mathit{takeWhile}, \mathit{dropWhile}, \mathit{span}, \mathit{break}, \\
\quad \mathit{lines}, \mathit{words}, \mathit{unlines}, \mathit{unwords}, \mathit{reverse}, \mathit{and}, \mathit{or}, \\
\quad \mathit{any}, \mathit{all}, \mathit{elem}, \mathit{notElem}, \mathit{lookup}, \\
\quad \mathit{sum}, \mathit{product}, \mathit{maximum}, \mathit{minimum}, \mathit{concatMap}, \\
\quad \mathit{zip}, \mathit{zip3}, \mathit{zipWith}, \mathit{zipWith3}, \mathit{unzip}, \mathit{unzip3}) \\
\quad \textbf{where} \\
\textbf{import}\ \mathit{qualified}\ \mathit{Char}\ (\mathit{isSpace}) \\
\textbf{infixl}\ 9\ !! \\
\textbf{infixr}\ 5\ +\!\!+ \\
\textbf{infix}\ 4\ \ \in, \notin
\end{array}
$$

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.

$$
\begin{array}{ll}
\mathit{head} & :: [a] \rightarrow a \\
\mathit{head}\ (x : \_) & = x \\
\mathit{head}\ [\,] & = \mathit{error}\ \texttt{"PreludeList.head: empty list"} \\
\mathit{last} & :: [a] \rightarrow a \\
\mathit{last}\ [x] & = x \\
\mathit{last}\ (\_ : \mathit{xs}) & = \mathit{last}\ \mathit{xs} \\
\mathit{last}\ [\,] & = \mathit{error}\ \texttt{"PreludeList.last: empty list"} \\
\mathit{tail} & :: [a] \rightarrow [a] \\
\mathit{tail}\ (\_ : \mathit{xs}) & = \mathit{xs} \\
\mathit{tail}\ [\,] & = \mathit{error}\ \texttt{"PreludeList.tail: empty list"}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{init} & :: [\,a\,] \to [\,a\,] \\
\mathit{init}\ [\,x\,] & = [\,] \\
\mathit{init}\ (x : \mathit{xs}) & = x : \mathit{init}\ \mathit{xs} \\
\mathit{init}\ [\,] & = \mathit{error}\ \texttt{"PreludeList.init: empty list"}
\end{array}
$$

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.

$$
\begin{array}{ll}
\mathit{null} & :: [\,a\,] \to \mathit{Bool} \\
\mathit{null}\ [\,] & = \mathit{True} \\
\mathit{null}\ (\_ : \_) & = \mathit{False}
\end{array}
$$

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 (!!):

$$
\begin{array}{lll}
(!!) & :: [\,a\,] \to \mathit{Int} \to a & \\
(x : \_)\ !!\ 0 = & & x \\
(\_ : \mathit{xs})\ !!\ n \mid n > 0 = & & \mathit{xs}\ !!\ (n-1) \\
(\_ : \_)\ !!\ \_ = & & \mathit{error}\ \texttt{"PreludeList.!!: negative index"} \\
[\,]\ \ !!\ \_ = & & \mathit{error}\ \texttt{"PreludeList.!!: index too large"}
\end{array}
$$

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 $n > \mathit{length}\ xs$. Similarly, *drop n xs* returns the suffix of *xs* after the first n elements, or $[\,]$ if $n > \mathit{length}\ xs$. Finally, *splitAt n xs* is equivalent to $(\mathit{take}\ n\ xs, \mathit{drop}\ n\ xs)$.

```
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.

```
takeWhile          :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x : xs)
    | p x          = x : takeWhile p xs
    | otherwise =  []

dropWhile          :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p xs@(x : xs')
    | p x          = dropWhile p xs'
    | otherwise =  xs

span, break        :: (a -> Bool) -> [a] -> ([a], [a])
span p []          = ([], [])
span p xs@(x : xs')
    | p x          = (x : xs', xs'') where (xs', xs'') = span p xs
    | otherwise =  (xs, [])

break p            = span (¬ ∘ p)
```

filter removes all elements not satisfying a predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x : xs) | p x = x : filter p xs
    | otherwise = filter p xs
```

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.

```
foldl              :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       =  z
foldl f z (x : xs) = foldl f (f z x) xs

foldl1             :: (a -> a -> a) -> [a] -> a
foldl1 f (x : xs) =  foldl f x xs
```

$$
\begin{array}{ll}
\mathit{foldl1}\ \_\ [\,] & = \mathit{error}\ \texttt{"PreludeList.foldl1: empty list"} \\
\mathit{foldr} & :: (a \to b \to b) \to b \to [a] \to b \\
\mathit{foldr}\ f\ z\ [\,] & = z \\
\mathit{foldr}\ f\ z\ (x : \mathit{xs}) & = f\ x\ (\mathit{foldr}\ f\ z\ \mathit{xs}) \\
\mathit{foldr1} & :: (a \to a \to a) \to [a] \to a \\
\mathit{foldr1}\ f\ [x] & = x \\
\mathit{foldr1}\ f\ (x : \mathit{xs}) & = f\ x\ (\mathit{foldr1}\ f\ \mathit{xs}) \\
\mathit{foldr1}\ \_\ [\,] & = \mathit{error}\ \texttt{"PreludeList.foldr1: empty list"}
\end{array}
$$

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:

$$\mathit{scanl}\ f\ z\ [x_1, x_2, ...] == [z, z\ \text{‘}f\text{‘}\ x_1, (z\ \text{‘}f\text{‘}\ x_1)\ \text{‘}f\text{‘}\ x_2, ...]$$

For example:

$$\mathit{scanl}\ (+)\ 0\ [1, 2, 3] \Rightarrow [0, 1, 3, 6]$$

Note that $\mathit{last}\ (\mathit{scanl}\ f\ z\ \mathit{xs}) = \mathit{foldl}\ f\ z\ \mathit{xs}$. *scanl1* is similar, but without the starting element:

$$\mathit{scanl1}\ f\ [x_1, x_2, ...] == [x_1, x_1\ \text{‘}f\text{‘}\ x_2, ...]$$

Here are the full definitions:

$$
\begin{array}{ll}
\mathit{scanl} & :: (a \to b \to a) \to a \to [b] \to [a] \\
\mathit{scanl}\ f\ q\ \mathit{xs} & = q : (\textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & \qquad [\,] \to [\,] \\
 & \qquad x : \mathit{xs} \to \mathit{scanl}\ f\ (f\ q\ x)\ \mathit{xs}) \\
\mathit{scanl1} & :: (a \to a \to a) \to [a] \to [a] \\
\mathit{scanl1}\ f\ (x : \mathit{xs}) & = \mathit{scanl}\ f\ x\ \mathit{xs} \\
\mathit{scanl1}\ \_\ [\,] & = \mathit{error}\ \texttt{"PreludeList.scanl1: empty list"} \\
\mathit{scanr} & :: (a \to b \to b) \to b \to [a] \to [b] \\
\mathit{scanr}\ f\ q0\ [\,] & = [q0] \\
\mathit{scanr}\ f\ q0\ (x : \mathit{xs}) & = f\ x\ q : \mathit{qs} \\
 & \qquad \textbf{where}\ \mathit{qs}@(q : \_) = \mathit{scanr}\ f\ q0\ \mathit{xs} \\
\mathit{scanr1} & :: (a \to a \to a) \to [a] \to [a] \\
\mathit{scanr1}\ f\ [x] & = [x] \\
\mathit{scanr1}\ f\ (x : \mathit{xs}) & = f\ x\ q : \mathit{qs} \\
 & \qquad \textbf{where}\ \mathit{qs}@(q : \_) = \mathit{scanr1}\ f\ \mathit{xs}
\end{array}
$$

$$scanr1\ \_\ [\,] \quad = error\ \texttt{"PreludeList.scanr1: empty list"}$$

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:

$$iterate\ f\ x \Rightarrow [x, f\ x, f\ (f\ x), ...]$$

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]

$$\begin{aligned}&iterate && :: (a \to a) \to a \to [a]\\ &iterate\ f\ x && = x : iterate\ f\ (f\ x)\end{aligned}$$

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.

$$\begin{aligned}&repeat && :: a \to [a]\\ &repeat\ x && = xs\ \mathbf{where}\ xs = x : xs\\[4pt] &replicate && :: Int \to a \to [a]\\ &replicate\ n\ x && = take\ n\ (repeat\ x)\\[4pt] &cycle && :: [a] \to [a]\\ &cycle\ [\,] && = error\ \texttt{"Prelude.cycle: empty list"}\\ &cycle\ xs && = xs'\ \mathbf{where}\ xs' = xs +\!\!+\ xs'\end{aligned}$$

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.)

$$
\begin{array}{ll}
\mathit{lines} & :: \mathit{String} \rightarrow [\mathit{String}] \\
\mathit{lines}\ \texttt{""} & = [\,] \\
\mathit{lines}\ s & = \mathbf{let}\ (l, s') = \mathit{break}\ (== \texttt{'\textbackslash n'})\ s \\
 & \qquad \mathbf{in}\ l : \mathbf{case}\ s'\ \mathbf{of} \\
 & \qquad\qquad [\,] \rightarrow [\,] \\
 & \qquad\qquad (\_ : s'') \rightarrow \mathit{lines}\ s''
\end{array}
$$

$$
\begin{array}{ll}
\mathit{words} & :: \mathit{String} \rightarrow [\mathit{String}] \\
\mathit{words}\ s & = \mathbf{case}\ \mathit{dropWhile}\ \mathit{Char.isSpace}\ s\ \mathbf{of} \\
 & \qquad \texttt{""} \rightarrow [\,] \\
 & \qquad s' \rightarrow w : \mathit{words}\ s'' \\
 & \qquad\quad \mathbf{where}\ (w, s'') = \mathit{break}\ \mathit{Char.isSpace}\ s'
\end{array}
$$

$$
\begin{array}{ll}
\mathit{unlines} & :: [\mathit{String}] \rightarrow \mathit{String} \\
\mathit{unlines} & = \mathit{concatMap}\ (+\!\!+ \texttt{"\textbackslash n"})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{unwords} & :: [\mathit{String}] \rightarrow \mathit{String} \\
\mathit{unwords}\ [\,] & = \texttt{""} \\
\mathit{unwords}\ \mathit{ws} & = \mathit{foldr1}\ (\lambda w\ s \rightarrow w +\!\!+ \texttt{' '} : s)\ \mathit{ws}
\end{array}
$$

reverse reverses the elements in a finite list.

$$
\begin{array}{l}
\mathit{reverse} :: [a] - [a] \\
\mathit{reverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]
\end{array}
$$

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.

$$
\begin{array}{ll}
\mathit{and}, \mathit{or} & :: [\mathit{Bool}] \rightarrow \mathit{Bool} \\
\mathit{and} & = \mathit{foldr}\ (\wedge)\ \mathit{True} \\
\mathit{or} & = \mathit{foldr}\ (\vee)\ \mathit{False}
\end{array}
$$

Applied to a predicate and a list, *any* determines if any element of the list satisfies the predicate. An analogous behavior holds for *all*.

$$
\begin{array}{ll}
\mathit{any}, \mathit{all} & :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow \mathit{Bool} \\
\mathit{any}\ p & = \mathit{or} \circ \mathit{map}\ p \\
\mathit{all}\ p & = \mathit{and} \circ \mathit{map}\ p
\end{array}
$$

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.

$$
\begin{array}{ll}
elem, notElem & :: (Eq\ a) \Rightarrow a \rightarrow [\,a\,] \rightarrow Bool \\
elem\ x & = any\ (== x) \\
notElem\ x & = all\ (\neq x)
\end{array}
$$

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.

$$
\begin{array}{l}
lookup \qquad\qquad :: (Eq\ a) \Rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b \\
lookup\ key\ [\,] \ = Nothing \\
lookup\ key\ ((x, y) : xys) \\
\quad \mid key == x = Just\ y \\
\quad \mid otherwise = lookup\ key\ xys
\end{array}
$$

A.10 Arithmetic on Lists

sum and *product* compute the sum and product, respectively, of a finite list of numbers.

$$
\begin{array}{ll}
sum, product & :: (Num\ a) \Rightarrow [\,a\,] \rightarrow a \\
sum & = foldl\ (+)\ 0 \\
product & = foldl\ (*)\ 1
\end{array}
$$

maximum and *minimum* return the maximum and minimum value, respectively from a non-empty, finite list whose element type is ordered.

$$
\begin{array}{l}
maximum, minimum :: (Ord\ a) \Rightarrow [\,a\,] \rightarrow a \\
maximum\ [\,] \ = error\ \texttt{"Prelude.maximum: empty list"} \\
maximum\ xs = foldl1\ max\ xs \\[4pt]
minimum\ [\,] \ = error\ \texttt{"Prelude.minimum: empty list"} \\
minimum\ xs = foldl1\ min\ xs
\end{array}
$$

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:

$$\begin{aligned}
&map :: (a \to b) \to [\,a\,] \to [\,a\,]\\
&map\ f\ [\,] = [\,]\\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}$$

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

concat appends together a list of lists:

$$\begin{aligned}
&concat :: [[\,a\,]] \to [\,a\,]\\
&concat\ xss = foldr\ (+\!\!+)\ [\,]\ xss
\end{aligned}$$

concatMap does what it says: it concatenates the result of mapping a function down a list.

$$\begin{aligned}
&concatMap \quad :: (a \to [\,b\,]) \to [\,a\,] \to [\,b\,]\\
&concatMap\ f = concat \circ map\ f
\end{aligned}$$

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.)

$$\begin{aligned}
&zip \quad :: [\,a\,] \to [\,b\,] \to [(a, b)]\\
&zip \quad = zipWith\ (,)
\end{aligned}$$

$$\begin{aligned}
&zip3 :: [\,a\,] \to [\,b\,] \to [\,c\,] \to [(a, b, c)]\\
&zip3 = zipWith3\ (,,)
\end{aligned}$$

Details: The functions *(,)* and *(,,)* are the pairing and tripling functions, respectively:

$$\begin{aligned}
&(,) \quad \Rightarrow \lambda x\ y \to (x, y)\\
&(,,) \Rightarrow \lambda x\ y\ z \to (x, y, z)
\end{aligned}$$

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.

$$
\begin{array}{lll}
\mathit{zipWith} & :: (a \to b \to c) \to [\,a\,] \to [\,b\,] \to [\,c\,] & \\
\multicolumn{3}{l}{\mathit{zipWith}\ z\ (a : as)\ (b : bs)} \\
 & = & z\ a\ b : \mathit{zipWith}\ z\ as\ bs \\
\mathit{zipWith}\ \_\ \_\ \_ & = & [\,] \\
\mathit{zipWith3} & :: (a \to b \to c \to d) \to [\,a\,] \to [\,b\,] \to [\,c\,] \to [\,d\,] & \\
\multicolumn{3}{l}{\mathit{zipWith3}\ z\ (a : as)\ (b : bs)\ (c : cs)} \\
 & = & z\ a\ b\ c : \mathit{zipWith3}\ z\ as\ bs\ cs \\
\multicolumn{3}{l}{\mathit{zipWith3}\ \_\ \_\ \_\ \_ = [\,]}
\end{array}
$$

The following two functions perform the inverse operations of *zip* and *zip3*, respectively.

$$
\begin{array}{ll}
\mathit{unzip} & :: [\,(a, b)\,] \to ([\,a\,], [\,b\,]) \\
\mathit{unzip} & = \mathit{foldr}\ (\lambda(a, b) {\sim} (as, bs) \to (a : as, b : bs))\ ([\,], [\,]) \\
\mathit{unzip3} & :: [\,(a, b, c)\,] \to ([\,a\,], [\,b\,], [\,c\,]) \\
\mathit{unzip3} & = \mathit{foldr}\ (\lambda(a, b, c) {\sim} (as, bs, cs) \to (a : as, b : bs, c : cs)) \\
 & \qquad ([\,], [\,], [\,])
\end{array}
$$

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 7, along with a simplified version of the class *Ord*. Here is its full specification of class *Ord*; note the many default methods.

$$
\begin{array}{l}
\textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where} \\
\quad compare\ \ :: a \rightarrow a \rightarrow Ordering \\
\quad (<), (\leqslant), (\geqslant), (>) :: a \rightarrow a \rightarrow Bool \\
\quad max, min :: a \rightarrow a \rightarrow a \\
\\
\quad compare\ x\ y \\
\qquad \begin{array}{lcl}
\mid x == y & = & EQ \\
\mid x \leqslant y & = & LT \\
\mid otherwise & = & GT
\end{array} \\
\\
\quad \begin{array}{lcl}
x \leqslant y & = & compare\ x\ y \neq GT \\
x < y & = & compare\ x\ y == LT \\
x \geqslant y & = & compare\ x\ y \neq LT \\
x > y & = & compare\ x\ y == GT
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\quad max\ x\ y \\
\qquad \mid x \geqslant y \qquad = \quad x \\
\qquad \mid otherwise = \quad y \\
\quad min\ x\ y \\
\qquad \mid x < \quad y \ = \quad x \\
\qquad \mid otherwise = \quad y \\
\mathbf{data}\ Ordering = LT \mid EQ \mid GT \\
\qquad \mathbf{deriving}\ (Eq, Ord, Enum, Read, Show, Bounded)
\end{array}
$$

Note that the default method for *compare* is defined in terms of ($\leqslant$), and that the default method for ($\leqslant$) 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*:

$$\mathbf{data}\ Color = Red \mid Orange \mid Yellow \mid Green \mid Blue \mid Indigo \mid Violet$$

can easily be given an *Enum* instance declaration, after which we can calculate the following results:

$$
\begin{array}{lll}
[Red\,..\,Violet] & \Longrightarrow & [\quad Red, Orange, Yellow, Green, \\
 & & \qquad Blue, Indigo, Violet] \\
[Red, Yellow\,..] & \Longrightarrow & [\quad Red, Yellow, Blue, Violet] \\
fromEnum\ Green & \Longrightarrow & 3 \\
toEnum\ 5 :: Color & \Longrightarrow & Indigo
\end{array}
$$

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:

$$
\begin{array}{lll}
\textbf{class } \mathit{Enum}\ a\ \textbf{where} & & \\
\quad \mathit{succ}, \mathit{pred} & :: a \to a & \\
\quad \mathit{toEnum} & :: \mathit{Int} \to a & \\
\quad \mathit{fromEnum} & :: a \to \mathit{Int} & \\
\quad \mathit{enumFrom} & :: a \to [a] & \text{-- [n..]} \\
\quad \mathit{enumFromThen} & :: a \to a \to [a] & \text{-- [n,n'..]} \\
\quad \mathit{enumFromTo} & :: a \to a \to [a] & \text{-- [n..m]} \\
\quad \mathit{enumFromThenTo} & :: a \to a \to a \to [a] & \text{-- [n,n'..m]}
\end{array}
$$

$$
\begin{array}{ll}
\quad\quad \text{-- Minimal complete definition: toEnum, fromEnum} & \\
\quad \mathit{succ} & = \mathit{toEnum} \circ (+1) \circ \mathit{fromEnum} \\
\quad \mathit{pred} & = \mathit{toEnum} \circ (\mathit{subtract}\ 1) \circ \mathit{fromEnum} \\
\quad \mathit{enumFrom}\ x & = \mathit{map}\ \mathit{toEnum}\ [\mathit{fromEnum}\ x\,..] \\
\quad \mathit{enumFromThen}\ x\ y & = \mathit{map}\ \mathit{toEnum}\ [\mathit{fromEnum}\ x, \mathit{fromEnum}\ y\,..] \\
\quad \mathit{enumFromTo}\ x\ y & = \mathit{map}\ \mathit{toEnum}\ [\mathit{fromEnum}\ x\,..\,\mathit{fromEnum}\ y] \\
\quad \mathit{enumFromThenTo}\ x\ y\ z = & \\
\quad\quad \mathit{map}\ \mathit{toEnum}\ [\mathit{fromEnum}\ x, \mathit{fromEnum}\ y\,..\,\mathit{fromEnum}\ z] &
\end{array}
$$

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:

$$
\begin{array}{rl}
\mathit{enumFrom}\ n & [n\,..] \\
\mathit{enumFromThen}\ n\ n' & [n, n'\,..] \\
\mathit{enumFromTo}\ n\ m & [n\,..\,m] \\
\mathit{enumFromThenTo}\ n\ n'\ m & [n, n'\,..\,m]
\end{array}
$$

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.

$$
\begin{array}{l}
\textbf{class } \mathit{Bounded}\ a\ \textbf{where} \\
\quad \mathit{minBound} :: a \\
\quad \mathit{maxBound} :: a
\end{array}
$$

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:

$$show :: (Show\ a) \Rightarrow a \rightarrow String$$

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:

$$\texttt{"The sum of "} +\!\!+\ show\ x +\!\!+ \texttt{" and "} +\!\!+\ show\ y +\!\!+ \texttt{" is "}$$
$$+\!\!+\ show\ (x + y) +\!\!+ \texttt{"."}$$

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:

$$showTree\ (Branch\ (Branch\ (Leaf\ 2)\ (Leaf\ 3))\ (Leaf\ 4))$$
$$\Longrightarrow \texttt{"< <2|3>|4>"}$$

We can define this behavior straightforwardly as follows:

$$showTree :: (Show\ a) \Rightarrow Tree\ a \rightarrow String$$

$$\begin{aligned}
&\mathit{showTree}\ (\mathit{Leaf}\ x)\\
&\quad = \mathit{show}\ x\\
&\mathit{showTree}\ (\mathit{Branch}\ l\ r)\\
&\quad = \texttt{"<"} \mathbin{+\!\!\!+} \mathit{showTree}\ l \mathbin{+\!\!\!+} \texttt{"|"} \mathbin{+\!\!\!+} \mathit{showTree}\ r \mathbin{+\!\!\!+} \texttt{">"}
\end{aligned}$$

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*:

$$\mathit{shows} :: (\mathit{Show}\ a) \Rightarrow a \rightarrow \mathit{String} \rightarrow \mathit{String}$$

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*:

$$\begin{aligned}
&\mathit{showsTree} :: (\mathit{Show}\ a) \Rightarrow \mathit{Tree}\ a \rightarrow \mathit{String} \rightarrow \mathit{String}\\
&\mathit{showsTree}\ (\mathit{Leaf}\ x)\ s\\
&\quad = \mathit{shows}\ x\ s\\
&\mathit{showsTree}\ (\mathit{Branch}\ l\ r)\ s\\
&\quad = \texttt{"<"} \mathbin{+\!\!\!+} \mathit{showsTree}\ l\ (\texttt{"|"} \mathbin{+\!\!\!+} \mathit{showsTree}\ r\ (\texttt{">"} \mathbin{+\!\!\!+} s))
\end{aligned}$$

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:

$$\mathit{showTree}\ t = \mathit{showsTree}\ t\ \texttt{""}$$

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):

$\textbf{type}\ \mathit{ShowS} = \mathit{String} \to \mathit{String}$

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:

$$
\begin{array}{l}
\mathit{showsTree} :: (\mathit{Show}\ a) \Rightarrow \mathit{Tree}\ a \to \mathit{ShowS} \\
\mathit{showsTree}\ (\mathit{Leaf}\ x) \\
\quad = \mathit{shows}\ x \\
\mathit{showsTree}\ (\mathit{Branch}\ l\ r) \\
\quad = (\texttt{"<"}+\!\!+) \circ \mathit{showsTree}\ l \circ (\texttt{"|"}+\!\!+) \circ \mathit{showsTree}\ r \circ (\texttt{">"}+\!\!+)
\end{array}
$$

Details: This can be simplified slightly more by noting that $(\texttt{"c"}+\!\!+)$ is equivalent to $(\texttt{'c'}:)$ 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 $(\texttt{"<"}+\!\!+)$ and $(\texttt{"a string"}+\!\!+)$ 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:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Show}\ a\ \textbf{where} \\
\quad \mathit{showsPrec} :: \mathit{Int} \to a \to \mathit{ShowS} \\
\quad \mathit{showList}\ \ :: [a] \to \mathit{ShowS} \\
\quad \mathit{showList}\ [\,] \qquad\quad = \mathit{showString}\ \texttt{"[]"} \\
\quad \mathit{showList}\ (x : \mathit{xs}) = \mathit{showChar}\ \texttt{'['} \circ \mathit{shows}\ x \circ \mathit{showl}\ \mathit{xs} \\
\qquad \textbf{where}\ \mathit{showl}\ [\,] \qquad\quad = \mathit{showChar}\ \texttt{']'} \\
\qquad\qquad\qquad \mathit{showl}\ (x : \mathit{xs}) = \mathit{showString}\ \texttt{", "} \circ \mathit{shows}\ x \circ \mathit{showl}\ \mathit{xs}
\end{array}
$$

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:

$$
\begin{aligned}
&\mathit{shows} :: (\mathit{Show}\ a) \Rightarrow a \rightarrow \mathit{ShowS}\\
&\mathit{shows} = \mathit{showsPrec}\ 0
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{show}\ \ :: (\mathit{Show}\ a) \Rightarrow a \rightarrow \mathit{String}\\
&\mathit{show}\ x = \mathit{shows}\ x\ \texttt{""}
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathbf{instance}\ (\mathit{Show}\ a) \Rightarrow \mathit{Show}\ (\mathit{Tree}\ a)\ \mathbf{where}\\
&\quad \mathit{showsPrec}\ n = \mathit{showsTree}
\end{aligned}
$$

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:

$$\mathbf{type}\ \mathit{ReadS}\ a = \mathit{String} \rightarrow [(a, \mathit{String})]$$

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]

$$\begin{aligned}
&\mathit{readsTree} :: (\mathit{Read}\ a) \Rightarrow \mathit{ReadS}\ (\mathit{Tree}\ a)\\
&\mathit{readsTree}\ (\texttt{'<'} : s) = [(\mathit{Branch}\ l\ r, u) \mid (l, \texttt{'|'} : t) \leftarrow \mathit{readsTree}\ s,\\
&\qquad\qquad\qquad\qquad\qquad (r, \texttt{'>'} : u) \leftarrow \mathit{readsTree}\ t]\\
&\mathit{readsTree}\ s = \qquad\quad [(\mathit{Leaf}\ x, t) \mid (x, t) \leftarrow \mathit{reads}\ s]
\end{aligned}$$

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 $\mathit{Branch}\ l\ r$ 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.:

$$\begin{aligned}
&(\mathit{reads}\ \texttt{"5 golden rings"}) :: [(\mathit{Int}, \mathit{String})]\\
&\Longrightarrow [(5, \texttt{" golden rings"})]
\end{aligned}$$

then you should be able to verify the following calculations:

$$\begin{aligned}
&\mathit{readsTree}\ \texttt{"<<1|2>|3>"}\\
&\Longrightarrow
\end{aligned}$$

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:

[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.

```
lex :: ReadS 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
```

$$
\begin{array}{l}
\quad\quad\quad\quad \textbf{where}\ \mathit{optional}\ r = g\ r \mathbin{+\!\!\!+} \mathit{mandatory}\ r \\
\quad\quad\quad\quad\quad\quad\quad\ \ \mathit{mandatory}\ r = [\,(x,u) \mid (\texttt{"("}, s) \leftarrow \mathit{lex}\ r, \\
\quad\quad\quad\quad\quad\quad\quad\ \ \mathit{sc} \quad\quad\quad\quad\quad\quad\quad\quad (x,t) \leftarrow \mathit{optional}\ s, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\texttt{")"}, u) \leftarrow \mathit{lex}\ t\,]
\end{array}
$$

The default method for *readList* is rather tedious, but otherwise straightforward.

reads can now be defined, along with an even higher-level function, *read*:

$$
\begin{array}{l}
\mathit{reads} :: (\mathit{Read}\ a) \Rightarrow \mathit{ReadS}\ a \\
\mathit{reads} = \mathit{readsPrec}\ 0 \\[4pt]
\mathit{read}\ \ :: (\mathit{Read}\ a) \Rightarrow \mathit{String} \rightarrow a \\
\mathit{read}\ s = \textbf{case}\ [\,x \mid (x,t) \leftarrow \mathit{reads}\ s, (\texttt{""}, \texttt{""}) \leftarrow \mathit{lex}\ t\,]\ \textbf{of} \\
\quad\quad\quad\quad [\,x\,] \rightarrow x \\
\quad\quad\quad\quad [\,] \rightarrow \mathit{error}\ \texttt{"PreludeText.read: no parse"} \\
\quad\quad\quad\quad \_ \rightarrow \mathit{error}\ \texttt{"PreludeText.read: ambiguous parse"}
\end{array}
$$

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 runtime 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 ($\mathit{read} \circ \mathit{show}$) 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:

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Ord}\ a) \Rightarrow \mathit{Ix}\ a\ \textbf{where} \\
\quad \mathit{range}\ \ \ \ :: (a,a) \rightarrow [\,a\,] \\
\quad \mathit{index}\ \ \ \ :: (a,a) \rightarrow a \rightarrow \mathit{Int} \\
\quad \mathit{inRange} :: (a,a) \rightarrow a \rightarrow \mathit{Bool}
\end{array}
$$

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,

$$\mathit{range}\ (0,4) \Longrightarrow [0,1,2,3,4]$$
$$\mathit{range}\ ((0,0),(1,2)) \Longrightarrow [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]$$

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:

$$\mathit{index}\ (1,9)\ 2 \Longrightarrow 1$$
$$\mathit{index}\ ((0,0),(1,2))\ (1,1) \Longrightarrow 4$$

B.7 The Numeric Classes

The *Num* class and the numeric class hierarchy were briefly described in Section 7.4. Figure B.1 gives the full class declarations.

class $(Eq\ a, Show\ a) \Rightarrow Num\ a$ **where**
  $(+), (-), (*) :: a \to a \to a$
  $negate :: a \to a$
  $abs, signum :: a \to a$
  $fromInteger :: Integer \to a$

class $(Num\ a, Ord\ a) \Rightarrow Real\ a$ **where**
  $toRational :: a \to Rational$

class $(Real\ a, Enum\ a) \Rightarrow Integral\ a$ **where**
  $quot, rem, div, mod :: a \to a \to a$
  $quotRem, divMod :: a \to a \to (a, a)$
  $toInteger :: a \to Integer$

class $(Num\ a) \Rightarrow Fractional\ a$ **where**
  $(/) :: a \to a \to a$
  $recip :: a \to a$
  $fromRational :: Rational \to a$

class $(Fractional\ a) \Rightarrow Floating\ a$ **where**
  $pi :: a$
  $exp, log, sqrt :: a \to a$
  $(**), logBase :: a \to a \to a$
  $sin, cos, tan :: a \to a$
  $asin, acos, atan :: a \to a$
  $sinh, cosh, tanh :: a \to a$
  $asinh, acosh, atanh :: a \to a$

class $(Real\ a, Fractional\ a) \Rightarrow RealFrac\ a$ **where**
  $properFraction :: (Integral\ b) \Rightarrow a \to (b, a)$
  $truncate, round :: (Integral\ b) \Rightarrow a \to b$
  $ceiling, floor :: (Integral\ b) \Rightarrow a \to b$

class $(RealFrac\ a, Floating\ a) \Rightarrow RealFloat\ a$ **where**
  $floatRadix :: a \to Integer$
  $floatDigits :: a \to Int$
  $floatRange :: a \to (Int, Int)$
  $decodeFloat :: a \to (Integer, Int)$
  $encodeFloat :: Integer \to Int \to a$
  $exponent :: a \to Int$
  $significand :: a \to a$
  $scaleFloat :: Int \to a \to a$
  $isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE$
    $:: a \to Bool$

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 $Int = (-2\hat{\ }29) \mid ... \mid -1 \mid 0 \mid 1 \mid ... \mid (2\hat{\ }29 - 1)$
data $Integer = \ ... -2 \mid -1 \mid 0 \mid 1 \mid 2 ...$

(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 $(a, b) \quad\quad = (a, b)$ -- more pseudo-code
data $(a, b, c) \quad = (a, b, c)$
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 $[a] = [\,] \mid a : [a]$ -- 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 $a \rightarrow [a] \rightarrow [a]$.

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 $(e_1, e_2, ..., en)$, $n \geqslant 2$, if Ti is the type of ei, then the type of the tuple is $(T_1, T_2, ..., Tn)$.

For $[e_1, e_2, ..., en]$,$n \geqslant 0$, 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:

$$\mathit{head}\ (x : \_) = x$$
$$\mathit{tail}\ (\_ : xs) = xs$$

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

[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.

what order are the matches attempted? What if none succeed? This section addresses these questions.

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 $[0, bot]$, then 1 fails to match 0, so the result is a failed match. But if $[1, 2]$ is matched against $[bot, 0]$, 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:

$$\begin{array}{lll} sign\ x \mid & x > 0 & = 1 \\ \quad\mid & x == 0 & = 0 \\ \quad\mid & x < 0 & = -1 \end{array}$$

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*:

$$\begin{array}{ll} take\ 0\ \_ & = [\,] \\ take\ \_\ [\,] & = [\,] \\ take\ n\ (x : xs) & = x : take\ (n - 1)\ xs \end{array}$$

and this slightly different version (the first 2 equations have been reversed):

$$\begin{array}{ll} take1\ \_\ [\,] & = [\,] \\ take1\ 0\ \_ & = [\,] \\ take1\ n\ (x : xs) & = x : take1\ (n - 1)\ xs \end{array}$$

Now note the following:

$$\begin{array}{lcl} \textit{take}\ 0\ \textit{bot} & \Longrightarrow & [\,] \\ \textit{take1}\ 0\ \textit{bot} & \Longrightarrow & \perp \\[1ex] \textit{take}\ \textit{bot}\ [\,] & \Longrightarrow & \perp \\ \textit{take1}\ \textit{bot}\ [\,] & \Longrightarrow & [\,] \end{array}$$

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:

$$f\ x_1\ x_2\ ...\ xk = \textbf{case}\ (x_1, ..., xk)\ \textbf{of}\ \begin{array}{l} (p_{11}, ..., p_{1k}) \rightarrow e_1 \\ ... \\ (p_{n1}, ..., p_{nk}) \rightarrow e_n \end{array}$$

where the *xi* are new identifiers. For example, the definition of *take* given earlier is equivalent to:

$$\begin{array}{l} \textit{take}\ m\ \textit{ys} = \textbf{case}\ (m, \textit{ys})\ \textbf{of} \\ \quad \begin{array}{ll} (0, \_) & \rightarrow [\,] \\ (\_, [\,]) & \rightarrow [\,] \\ (n, x : xs) & \rightarrow x : \textit{take}\ (n-1)\ xs \end{array} \end{array}$$

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

[Ass13a] MIDI Manufacturers Association. Complete midi 1.0 detailed specification, 1995-2013.

[Ass13b] MIDI Manufacturers Association. General midi 1, 2 and lite specifications, 1995-2013.

[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.

[Cag86] John Cage. *Silence: Lectures and Writings*. Wesleyan University Press, Middletown, CT, 1961,1986.

[CE01] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proc. of the 2001 Haskell Workshop*, September 2001.

[Cho73] John M. Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of Audio Engineering Society*, 21(7):526–534, 1973.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ, 1941.

[Coo02] Perry Cook. *Real Sound Synthesis for Interactive Applications*. A.K. Peters Press, Natick MA, USA, 2002.

[Cor94] Chick Corea. *Children's Songs – 20 Pieces for Keyboard (ED 7254)*. Schott, Mainz, 1994.

[Cou04] Antony Courtney. *Modelling User Interfaces in a Functional Language*. PhD thesis, Department of Computer Science, Yale University, May 2004.

[EH97] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming, Oxford University*. Springer Verlag, LNCS 2638, 2003.

[Hin69] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[HMGW96] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[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.

[Hud96] Paul Hudak. Haskore music tutorial. In *Second International School on Advanced Functional Programming*, pages 38–68. Springer Verlag, LNCS 1129, August 1996.

[Hud00] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[Hud03] Paul Hudak. Describing and interpreting music in Haskell. In *The Fun of Programming*, chapter 4. Palgrave, 2003.

[Hud04] Paul Hudak. An algebraic theory of polymorphic temporal media. In *Proceedings of PADL'04: 6th International Workshop on Practical Aspects of Declarative Languages*, pages 1–15. Springer Verlag LNCS 3057, June 2004.

[Hug00] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[KS83] Kevin Karplus and Alex Strong. Digital synthesis of plucked string and drum timbres. *Computer Music Journal*, 7(2):43–55, 1983.

[LH07] Paul Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, November 2007.

[Mil78] R.A. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

[Moo90] F. Richard Moore. *Elements of Computer Music*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[Nel95] Gary Lee Nelson. Bifurcate me, baby!, 1995.

[P$^+$03] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[Pat01] Ross Paterson. A new notation for arrows. In *ICFP'01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.

[Pie91] B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, MA, 1991.

[PJW93] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993. 71–84.

[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.

[She64] Roger N. Shepard. Circularity in judgements of relative pitch. *Journal of the Acoustical Society of America*, 36(12):2346–2353, December 1964.

[Ver86] B. Vercoe. Csound: A manual for the audio processing system and supporting programs. Technical report, MIT Media Lab, 1986.

[Wad92] P. Wadler. The essence of functional programming. In *Proceedings 19th Symposium on Principles of Programming Languages*, pages 1–14. ACM, January 1992.

[WCH12] Daniel Winograd-Cort and Paul Hudak. Wormholes: Introducing Effects to FRP. In *Haskell Symposium*, pages 91–103. ACM, September 2012.

[WCLH11] Daniel Winograd-Cort, Hai Liu, and Paul Hudak. Virtualizing Real-World Objects in FRP. Technical Report YALEU/DCS/RR-1446, Yale University, July 2011.