Perspectives on Computer Hardware and Software: Logic in Action

Rex Page Ruben Gamboa

September 17, 2010

Contents

Part I

Introduction to Computers and Logic

Chapter 1

Computer Systems: Complex Behavior from Simple Principles

Computer systems, both hardware and software, are some of the most complicated artifacts that humans have ever created. But at their very core, computer systems are straightforward applications of the same logical principles that philosophers have been developing for more than two thousand years, and this book will show you how.

The hardware component of a computer system is made up of the visible pieces of equipment that you are familiar with, such as monitors, keyboards, printers, web cameras, and USB storage devices. It also includes the components inside the computer, such as chips, cables, hard drives, DVD drives, and boards. The properties of hardware devices are largely fixed when the system is constructed. For example, a hard drive can store 2 gigabytes, or a cable may be able to transmit 25 different signals simultaneously—but no more.

The hardware that makes up a computer system is not much different than the electronics inside a television set or DVD player. But computer hardware can do something that other consumer electronic devices cannot; it can respond to information encoded in a special way, what is referred to as the software component of the system. We usually call the software a "program."

You may have heard about different computer chips and their instruction sets, such as the Intel chips that power the laptop we are using to write

this book. Software for these chips consists of a list of instructions, and that is the way that most general-purpose computer systems in use today represent software. But there is nothing magical about this way of thinking about software; in fact, we believe that thinking about software in this way obscures its important features.

For example, during the 1980s and 1990s, special computers were created that used mathematical functions as the basis for the software. And other computer systems have completely different representations for software. The Internet service Second Life, for example, can be programmed using Scratch for Second Life (S4SL), and programs in S4SL look like drawings, not instructions. An even more graphical view of software is provided by LabView, which is used by scientists and engineers all over the world to control laboratory equipment. If you are at a university, there is a good chance that LabView is being used at one of the laboratories near you! And software in LabView looks like a large engineering drawing.

In this book, we present software as a collection of mathematical equations. This view is entirely different from, yet consistent with and equivalent to, the view of software as a list of instructions or a specialized drawings. The most important advantage of equations is that they make computer software accessible to anyone who understands high school algebra.

It is the software that gives a computer system much of its power and flexibility. An iPhone, for example, has a screen that can display 441,600 different pixels arranged in a 960x460 grid, but it is the software that determines whether the iPhone displays an album cover or a weather update. Software makes the hardware more useful by extending its range of behavior. For instance, the an iPhone audio device may be able to produce only a single tone at a time. But the software can instruct it to play a sequence of tones, one after another, that sound like Beethoven's Fifth Symphony.

Think of the hardware as the parts in a computer that you can see, and the software as information that tells the computer what to do. But the distinction between hardware and software is not as clear cut as this suggests. Many hardware components actually encode software directly and control other pieces of the system. In fact, hardware today is designed and built using many techniques first developed to build large software projects. And a major theme of this book is that both hardware and software are realizations of formal logic.

The distinction between hardware and software is well and good, but it leaves many questions to the imagination, such as:

Logicians and mathematicians have been studying models of computation since before computers were invented. This was done, in part, to answer a deep mathematical question: What parts of mathematics can, in principle, be fully automated? In particular, is it possible to build a machine that can discover all mathematical truths?

As a result, many different models of computation were developed, including Turing machines, Lambda calculus, partial recursive functions, unrestricted grammars, Post production rules, random-access machines, and many others. Historically, computer science theory has treated Turing machines as the canonical foundation for computation, the random-access model more accurately describes modern computers. The equational model of computation that we use in this book falls into the to the Lambda calculus bailiwick. What is truly remarkable is that all of these different models of computation are equivalent. That is, any computation that can be described in one of the models can also be described in any of the other models. An extension of this observation conjectures that all realizable models of computation are equivalent. This conjecture is known as the Church-Turing Thesis, and it lies at the heart of computer science theory.

Another remarkable fact is that some problems cannot be solved by a program written in any of these computational models. Alan Turing, who many consider to be the first theoretical computer scientist, was the first to discover such uncomputable problems, shortly after the logician Kurt Gödel showed that no formal system of logic could prove all mathematical truths. Turing's and Gödel's discoveries of incomputability and incompleteness show that it is not possible to build a machine that can discover all mathematical truths, not even in principle.

Aside 1: Models of Computation

1. How can software affect hardware? Example: Instruct an audio device to emit a sound.

2. How can software detect the hardware's status? Example: Determine whether a switch is pressed or not.

3. What is the range of instructions that software should be able give hardware? Examples: Add two numbers. Select between two formulas, depending on the results of other computations. Replace one formula by another.

We'll start with the question about the range of instructions. Each model of computation (see Aside 1) provides an answer to this question. There are as many answers as there are models of computation, and logicians have been very creative when it comes to constructing such models! Luckily, these models are completely equivalent to each other, so we can choose the answer that is most convenient to us. And the most convenient answer is that a program consists of basic mathematical primitives, such as the the basic operations of arithmetic (addition, multiplication, etc) and the ability to define new operations based on previously defined operations.

Once we take basic arithmetic functions (that is "operation") and the ability to define new ones as the basic model of computation, we can talk about what it is possible for software to do. Software can affect hardware by the values that a function delivers. For example, a program—that is, a function—can tell an iPhone what to display in its screen by delivering a 960x460 matrix of pixels. Each entry in the matrix can be a number that represents a color, for example 16,711,680 for red or 65,280 for green. Similarly, the hardware can inform the software of the status of a component by triggering a function defined in the software and supplying the status in the parameters of the function. For example, the parameters of a particular function could be the coordinates of the pixel selected by touching the screen. Other gestures, such as tapping or scrolling, would trigger different operations.

Let's consider a simple example. We will build a simple computer device that plays rock-paper-scissors. The machine has three buttons, allowing the user to select rock, paper, or scissors. The device also has a simple display unit. After the user makes a selection, the display unit shows the computer's choice and lets the user know who won that round. Our program will make the selection and determine the winner. To keep things fair, we will write

We will use the terms "operation" and "function" interchangeably. Some models of computation use these terms to mean different things, but in the model we will use makes it convenient to think of them as the same thing. So, when we say "function" we mean "operation" and vice versa. And what we're talking about when we use either term is a transformation that delivers results when supplied with input. We refer to the results delivered by a function (or operation) as its "value", and we refer to the input supplied to the function as the parameters of the function. Sometimes we use the term "argument" instead of "parameter", but we mean the same thing in either case. When talking about the input to an "operation", we often use the term "operand", but we could also use either of the other terms for input, "parameter" or "argument", since they all mean the same thing in our model.

To summarize, a function (aka, operation) is supplied with parameters (aka, arguments, operands) and delivers a value. A formula like $f(x, y)$ denotes the value delivered by the function f when supplied with parameters x and y. For example, if f were the arithmetic operation of addition, then $f(x, y)$ would stand for the value $x + y$, $f(2, 2)$ would denote 4, and $f(3, 7)$ would stand for 10.

Aside 2: Operations, Functions, Operands, Parameters, and Arguments

this program as two separate functions, so the machine cannot "see" the human player's choice.

The first function, called `emily`[1], makes the computer's choice. This function will deliver either "rock" or "paper" or "scissors" as its value. We could use 0, 1, and 2 as shorthand for these values, but computers are versatile enough with any type of information, not just numbers, so we are going to stick with the longer names, to make it easier for us to keep track of what things mean.

But what about the input to the function `emily`? The value of a mathematical function is completely determined by its inputs. That's what it means to be a function! So if `emily` has no inputs, it must always return the same value, and that would be a very boring game. We've already decided that it would be unfair for `emily` to see the player's choice, but what about the last round? It is fair for the machine to makes its selection by

[1]This function is named after a child of one of the authors. The "real" Emily plays rock-paper-scissors game just like the program developed in this chapter.

considering the previous round of the game, so the input can be the user's *previous* choice—or a special token, like "N/A" for the game's first round. The function `emily` can now be described as follows:

$$emily(u) = \begin{cases} \text{“rock”} & \text{if } u = \text{“scissors”} \\ \text{“paper”} & \text{if } u = \text{“rock”} \\ \text{“scissors”} & \text{otherwise} \end{cases}$$

The second function, which can be called `score`, decides who was the winner of the round. Its input corresponds to the choices made by the computer and the user, respectively. Its output consists of a pair or values. The first value determines the winner, and the second "remembers" the user's choice.

$$score(c, u) = \begin{cases} (\text{“none”}, u) & \text{if } c = u \\ (\text{“computer”}, u) & \text{if } (c, u) = (\text{“rock”}, \text{“scissors”}) \\ (\text{“user”}, u) & \text{if } (c, u) = (\text{“rock”}, \text{“paper”}) \\ (\text{“computer”}, u) & \text{if } (c, u) = (\text{“paper”}, \text{“rock”}) \\ (\text{“user”}, u) & \text{if } (c, u) = (\text{“paper”}, \text{“scissors”}) \\ (\text{“computer”}, u) & \text{if } (c, u) = (\text{“scissors”}, \text{“paper”}) \\ (\text{“user”}, u) & \text{if } (c, u) = (\text{“scissors”}, \text{“rock”}) \end{cases}$$

What is the point of the second value of `score`? As you can see, it is always equal to u, i.e., the user's selection. The intent is that this second returned value will be passed as the input to the next call of `emily`. This is how the program can remember the user's previous choice.

We could certainly have dropped this second return value, and simply stipulated that the hardware is responsible for remembering the user's last selection. However, we chose to write the program in this way for two reasons. First, it gives us more flexibility. The hardware is simply responsible for storing the value returned by `score` and sending it to `emily` in the next round. We can make the program a more sophisticated player of rock-paper-scissors by changing the software, possibly changing the value that is passed from `score` to `emily` in the process. For example, a more sophisticated program may want to keep track of the number of times that the user has selected each of the choices. Since it's the software that decides what to remember from each round, this choice can be changed very easily. That is the flexibility and power of software.

The second reason for writing the program in this way is that it gives us an opportunity to make an important point about our computational

model. Since programs in our model consist of collections of equations defining mathematical functions, they cannot "remember" anything. This will come as a surprise to programmers who are used to other computational models (such as C++ or Java). Those models to store values in "variables" and use those variables later in the program. We cannot do that in our purely functional model. If we did, we would lose the ability to understand our programs in terms of classical logic and would have to move into and unfamiliar and much more complicated domain.

Fortunately, our model allows the *hardware* to keep track of certain values, in much the same way that the hardware can tell which button has been pressed. Our model of computation simply allows the hardware to have some "hidden" buttons that can store information and later pass it along to a function in the form of a parameter.

This model is simple, but has as much power to specify computations as any other model. It is reasonable to believe, based on what they can do, that computers must be much more complicated than that. But in fact, that's all there is to. Power from simplicity, a bargain if there ever was one.

Consider *Deep Blue*, the computer that beat world champion Gary Kasparov at chess on May 11, 1997. This program can be written as a function whose input is an 8x8 matrix of numbers that represent the position of the pieces on the board after Kasparov's last move. The function's output is either the position of the board after its move, or a special white-flag token used to "resign" the game.

In principle, a chess-playing function can be written as follows. Given the input board, determine all possible legal moves. If there are no legal moves, resign. If there is a legal move that results in checkmate, do that move. Otherwise, for each legal move, consider each of the possible moves by the opponent. Each one of those results in a new board, which can then be examined by our chess-playing function!

It may seem surprising that a function can be defined this way, but circular definitions of functions are common in mathematics. We usually refer to them as "inductive", but "circular" is just as good. The trick is that the circularity, that is the place in the definition that refers to the function being defined, represents a reduced level of computation. Some parts of the definition will not be circular, and any circular reference will involve parameters that are closer to a non-circular portion of the definition than the parameters supplied to that portion of the definition.

Functions that have circular (that is, inductive) definitions are some-

times called "recursive" functions. We try to avoid that term because it is often associated with specialized ways to carry out the computation that the function describes, but we may slip up from time to time and use the term "recursive".

For example, the "function" that adds the first five reciprocals of the natural numbers can be written as

$$reciprocals\_5 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$$

This is obviously not an inductive definition. For that matter, this "function" is really a constant, not a function in the common sense of the word. But what about a function that computes the sum of the first N reciprocals? That is where we need to use an inductive definition. The key idea is that this sum is simply $\frac{1}{N}$ plus the sum of the first $N-1$ reciprocals. More formally, it is defined by the following equation

$$reciprocals(n) = \begin{cases} 0 & \text{if } n = 0 \\ reciprocals(n-1) + \frac{1}{n} & \text{otherwise} \end{cases}$$

As you can see, the definition of reciprocals is circular, but the parameter in the circular part is closer to zero than the parameter that arrived in the first place. And the definition is not circular when the parameter is zero. So, you can use the definition to compute the value of $reciprocals(5)$ by computing $reciprocals(4)$, which is computed by first computing $43ciprocals(3)$, and so on down the line. Eventually, you come to $reciprocals(0)$, for which the definition supplies the value one in non-circular fashion. Then, you can work your way back up the line and arrive that conclusion that $reciprocals(5) = \frac{137}{60} \approx 2.3$.

Surely it is surprising that all properties of a function can be derived from a definition that covers only a few special cases, and with circularity in the mix to boot. But that is the way it is. Every computable function has a definition of this form, and that is the way we will define functions throughout this book.

To get back to the chess-playing function, the problem with our naive approach is that there are too many moves to consider. While the function can, *in principle*, select the best move to play next, *in practice*, the computation would take too much time. The sun would be long dead before the computer could decide on its first move. But in fact, *Deep Blue* did play like this, only it used a massively parallel computer so that it could consider

many moves at the same time, and in most cases it considered only six to eight moves in the future. In practice, *Deep Blue* is a complicated function, with a large definition. But in principle, it is just a mathematical function, just like the programs we will study in this book.

Ex. 1 — What happens if you try to compute $reciprocals(-1)$? How about $reciprocals(\frac{1}{2})$?

Ex. 2 — Define a function called $factors(k, n)$ that returns true if and only if k is a factor of n. You may assume that the function $mod(k, n)$, which returns the remainder of k/n, has already been written.

Ex. 3 — Define a function called $largest\_factor(n)$ that returns the largest factor of n, other than n itself. For example, $largest\_factor(30) = 15$ and $largest\_factor(15) = 5$. *Hint:* You may find it necessary to start with the function $largest\_factor\_upto(n, k)$ that finds the largest factor of n, but no larger than k.

Ex. 4 — Modify the definition of the function $reciprocals(n)$ so that it adds the reciprocals only of the primes less than or equal to n. You can use the function $largest\_factor(n)$ in your solution. Impress your friends with useless trivia: While the original function $reciprocals(n)$ grows without bound as n becomes large, the modified version that only adds primes is bounded.