Keeping big things simple

David Honour

May 10, 2016

I am going to tell you the things I wish someone had told me. Many of my colleagues think I'm mad for trying this as much of what I will talk about is very abstract. Much like the idea of π as the ratio of the diameter to the circumference of a circle is useful, it can't be used for calculations. However numerical approximations of π are applicable in many situations. Similarly manifestations in code of some of the ideas here form the toolkit of practical programming.

1 Big little ideas

The things introduced in this section are small ideas with big implications. I don't think any of them is complex, the difficulty is in remaining aware of them when deep in the details of the problem you're trying to solve.

However, perhaps counter intuitively doing so can actually help to keep your mind on what you are trying to achieve in the longer term. This is mainly because the things you make will often be simpler and easier to understand.

1.1 Data

In spoken english words are sequences of sounds that, in and of themselves these sounds don't mean anything. We ascribe them meaning in order to communicate concepts.

Computer memory is essentially boxes with patterns in. Like the sounds that make up speech the patterns don't mean anything on their own and must be interpreted as a representation of something else.

There are some words (for instance their, there and they're) that sound indistinguishable, but have different meanings. This is true in these memory patterns too. Therefore each pattern also has an implicit meaning (or type)[1].

If each box is assigned an addresses then you can have a value that refers to a piece of memory, this is called a pointer. Table 1 contains some examples of these patterns and their possible meanings.

[1]data like this about data is often called metadata

Address	Pattern	What it represents	Value
0x00	0x48	unsigned	72
0x01	0xFF	signed	-1
0x02	0x48	char	H
0x03	0x02	char *	char at 0x02

Table 1: Some example bit patterns and their meanings. Note that the right two headings aren't actually stored, the programmer must keep track of what the data means.

1.2 Operations

Processors implement operations that transform data, for instance by doing arithmetic. These operations have patterns assigned to them so you can select which one to do:

Pattern	Operation
0xA0	add
0xA1	subtract

You can make arrays of these patterns, called operation codes or opcodes. These determine what a computer executes, it's just data.

1.3 Side effects

A side effect is where a procedure changes some state or has an observable interaction with the rest of the system.

Assume we have an instruction that adds two unsigneds "in place"[2]. If we have a list of 3 unsigned numbers and we want the sum of them what might we do?

We could add numbers 1 and 2 to the 0th, then the 0th number would be the sum. No memory above that required to store the numbers is needed and it has been done in the minimum possible number of operations. A side effect has occurred, the original data has been changed.

Assume we have in place instructions for $+ - \div \times$. Can we find the mean, μ, of 3 numbers?

$$\mu = \frac{a+b+c}{3}$$

How about the variance V?

$$V = \frac{(a-\mu)^2 + (b-\mu)^2 + (c-\mu)^2}{3}$$

Can we do each step with what we have? Can we do the whole operation with the instructions we have? How about if we introduce a copy instruction?

[2]such that the result is stored in the place where the input used to be

1.4 Immutability

Even in this simple example keeping track of what data is about to be changed (or mutated) is getting a bit fiddly. A proceedure operating on immutable data may not change its input.

This is useful because it makes it much simpler to assemble larger code from building blocks that behave in this way as they don't "trample" the input. This also means that if you give 2 procedures the same piece of data they can't couple together (thus introducing complexity) with it.

Have another go at the variance, this time the results of the arithmetic instructions go into a 3rd piece of memory leaving the inputs untouched.

1.5 Branches

So far we have talked about instructions as though they are a linear list, read from top to bottom, but haven't said what keeps track of where we are. There is a special piece of memory, called the instruction pointer, which identifies which instruction to do next. At the start of the processing cycle this pointer is followed to find the next instruction to perform. This is usually incremented[3] at the end of each instruction so that we move through the instructions as if we were reading them.

There are some instructions (called jumps) that can change the value of the instruction pointer (and therefore change which instructions are next). Try making a counter with in place addition and a simple jump.

We would also like to be able to do different things depending on data we get whilst running. This "changing direction" based on input is known as branching. So we use a conditional jump operation, for instance if this is 0 jump here, otherwise here to change the flow of the program based on this data. Try making a counter that stops at 5 with this.

1.6 Encapsulation

Encapsulation is wrapping up a series of steps (or ideas) into something more semantic (meaningful). This is used to lower cognitive load (amount of stuff to think about simultaneously).

Say I ask you to make some tea. What do you actually have to do to make that happen?

1. Boil water.
2. Add bag and water to cup.
3. Wait.
4. Remove bag.

[3]has one added to it

Consider one step of this, what does doing that actually involve?

Each time we break things down more we get closer to a specific set of instructions that can be followed to achieve our aim. However you wouldn't want to express how to prepare an entire meal in the lowest level instructions. It would be far too cumbersome, and if you were to be given them would you be able to tell what they were trying to acheive?

1.7 Testing

Whenever you try something to see if it works, that's testing. Say you were making a program that sums a list of numbers. How can we tell if it works?

We can pick an example and check that the result is what we expect. For instance in our example we know that:

$$3 + 4 + 5 == 12$$

so we could manually enter those numbers and see if it works. Having to do that to see if it still works every time the code is changed gets annoying really fast, so we can write some code to test our function:

$$\mathrm{sum}(3, 4, 5) == 12$$

this is $true$ if the function works and $false$ if it doesn't, so we can run this to tell us if the sum program works as expected.

In this example it is somewhat excessive to do this, but as larger and larger problems are tackled knowing that all the bits work even if they have been changed becomes increasingly important in working out what's broken. Customers don't like being sent new versions that fix some things and break others so this kind of testing is extremely important in retaining them.

If we want to test our program fully we must follow all branches (otherwise that code will not be run and we won't know if it all works). If the branches are independant (e.g. they don't interact) then the number of tests goes as $2n_{\text{branches}}$. However if all our branches interact the number of tests we require goes as $n_{\text{tests}} = 2^{n_{\text{branches}}}$ which quickly becomes impractically large.

Often we have a mixture of interacting and non-interacting branches, so the second term dominates. Hence we often test components at each level covering all of their branches, before moving to the next[4].

2 Language Paradigms

In principle you could write everything as lists of opcodes. That's what your computer actually runs, on some level everything is. However if you think back to the tea example, you can see that writing anything large in that way will quickly become incomprehensible. So let's talk about some constructs that are used to ease the cognitive load.

[4]this is usually called unit testing

2.1 Procedural

You might note what we have been talking about so far results in highly repetitive code. For instance we might have liked to use the code we already had for the sum when finding the mean?

2.1.1 Macros

Macros let us name chunks of operations for reuse in a parameterised way. They're like templates, they may have spaces that we fill in when we use them.

Write some instructions (high level) for making coffee. You can write instructions for a person making tea + coffee by writing a "make a hot drink" macro and substituting the parts that are drink type specific.

So now we have a reusable block, which is great in that you don't have to type it repeatedly. It also has an additional, perhaps less obvious benefit in that if there's a problem with it we are more likely to find it, as we use it more, and only have to fix it once.[5]

As good as this reduction in repetition is, there are still some irritations. We still need to know about the contents of every macro, for instance where it stores its output, in order to use it. Our code also becomes very large as everywhere we use a macro, the instructions are repeated.

2.1.2 Functions

Functions formalise how input, output and control flow around a chunk of instructions work. What properties might we want from rules about these things?

We'd like to know what values our inputs have taken, without knowing what they're called. So we supply those inputs in an array in a known location.

We'd like to know where the result is. So we pass into every set of instructions where we would like the result to go.

We'd like to be able to reuse the chunk without copying it, this means we need it to know where to go next. So we pass in a value for the instruction pointer we want the code to go afterwards.

We might also like to have somewhere to put intermediate results. This means our instructions themselves can be reused, not just the templates of them. A mechanism[6] is often provided for this.

With this in place we no longer have to know about the details of each chunk of code we use, only its intent and standardised function boundary (known as its function signature).

2.2 Object Oriented

Consider for instance our 2 integer representations, signed and unsigned. If you want to add them then there are 4 operations for this, one for each combination of the types.

[5] code exhibiting little repetition is often referred to as DRY (Don't Repeat Yourself)

[6] usually a stack

Say you just want to add them and you don't want to have to know what type they are. What approaches to solve this can you think of? What advantages and disadvantages do they have?

2.2.1 Classes

Objects stick functions to data so you don't have to know which one to call. These objects are usually called classes.

For our integer addition example, the class might look something like this:

"Type"	Add unsigned	Add signed	Value
Unsigned	add_unsigned_unsigned	add_unsigned_signed	12
Signed	add_signed_unsigned	add_signed_unsigned	-3

2.2.2 Polymorphism

As a user of a class, we don't have to know which objects we operate on anymore. So long as they have the structure we expect, we can add and remove new types at any time without changing the calling code.

For example if we have a set of classes representing animals that provide a "say" method. We can write some code that works for all animals that tells you what they say, without knowing what the animals are.

2.3 Functional

We've seen how side effects, and mutable data can cause us trouble. So what happens if we constrain ourselves not to have them?

2.3.1 Recursion

How do you loop when you can't change the value you assigned to something? You can call your function from itself:

$$sum(a) = a;\ sum(a, ...) = a + sum(...)$$

2.3.2 Metafunctions

Say we wanted to multiply instead of add a list of integers. How could we do that? Write a recursive version of the product function in the same way as we did for sum above.

However having to write recursive versions of every way of combining 2 things we could ever want to do is going to get annoying really fast.

What if we pass one function to another? We could, for example write a function that applies any 2 input function to reduce a list down to a single value. This function is usually called fold or reduce:

$$\text{fold}(f, i, x) : f(i, x);\ \text{fold}(f, i, x, ...) = f(x, \text{fold}(f, i, ...))$$

where f is the 2 input function to apply, i is an initial value to start the process with and ... is a list.

2.3.3 Dynamic functions

Think of a function that adds 3 to another number:

$$f(x) : x + 3$$

That's a function you can write by hand, say I want one of those for each integer, you could be writing them out for some time.

We can use functions can define new functions. For our example we need a function that takes in a number and returns a function that adds the number we asked to be added:

$$g(y) : (f(x) : x + y)$$

2.4 "Faux-O"

We're now going somewhere that many professionals haven't considered. What do we get if we do all the things we've talked about at once? This paradigm doesn't have a widely recognised name as the others do, so this is less googleable[7].

Given that they cannot mutate the instance, what do the methods on these classes do? Return new instances, constructed such that the desired changes are made.

Try implementing addition with this.

3 System Paradigms

Programs are often used together to make something that works, these composite entities are called systems.

3.1 Batch Jobs

Think of a production line, each component is given the result of the previous one, and its result is passed along to the next.

Let's consider a scoreboard. We could have a button that sends a signal to a counter which sends a total on to a display.

The components are known and fixed. If in our example I wanted to add a second display the system has to be changed.

3.2 Remote Procedure Calls

A web browser displays data that it requests from servers, forming a 2 component system. The browser doesn't know about every site it might be asked to connect to, and the servers don't know who might connect to them.

[7]the name used here is from a Gary Bernhardt's Boundaries talk.

We could do a score card here too. We could have a page used to update the score, and one for viewing it. The one to update the score can only be used by one person at once.

You have to refresh the view page frequently to keep up to date. This repeated data getting to see if it has changed is called polling.

3.3 Queues

Let's say we introduce queues, messages can be put onto them, or consumed from them.

How would we do the score board here? Button presses put something in an "input" queue. The counter consumes the "input" queue putting the latest total on an "output" queue. The display consumes the "output" queue displaying the latest value.

Which components need to know about which others? The only thing everyone has to know about is where the queues are.

What happens if we add a second display to this?

4 If this intrigued you...

Look up Gary Bernhardt, his screencasts and conference talks are great. Other than that, I'm happy to point you to things based on what interests you. My email is davhonou@cisco.com if you want to take that offer up.

If I get around to fixing any mistakes or updating this then the latest version of this can be found at: `https://github.com/foolswood/ideas_and_paradigms_talk/raw/master/oo.pdf`.