Advanced Programming Techniques

John Skaller

November 1, 2021

Contents

Chapter 1

Review

Here we are going to review some basic concepts you should already know.

1.1 Invariants

Definition 1. An *invariant* is a quantity which does not change.

1.1.1 Loop Invariants

Invariant Variable

Consider this code snippet:

```
for(int i=0; i<100; ++i) {
  int j  = 1;
  int k = j + i;
  cout << k << endl;
}
```

Here the variable j has the same value 1, no matter which iteration of the loop is being executed. So we can say j is a loop invariant. There is an optimisation known as *invariant code motion* which allows us to rewrite the code to be more efficient without changing the semantics, by moving the invariant out of the loop body:

```
int j  = 1;
for(int i=0; i<100; ++i) {
  int k = j + i;
  cout << k << endl;
}
```

So an advantage of recognising invariants is that doing so can lead to better performance by *optimisation*.

In this case we can make a further modification:

```
      int const j = 1;
```

which probably has no impact on performance of the code but has a significant impact on ability to reason about the code. We are using a language feature which assures us that the variable `j` is indeed invariant.

Invariant Condition

Now consider this code:

```
    int j  = 100;
    for(int i=0; i<100; ++i, --j {
      assert(j + i == 100);
      cout << j << "," << i << endl;
    }
```

Here, the loop invariant is that $j+i$ is always 100 inside the loop, in every pass. This is checked by the `assert` statement. We believe the invariant because, initially, $i = 0$ and $j = 100$ so $i + j = 100$ and every iteration, i is incremented by 1, but j is decremented by 1, and so the change in the sum is the sum of the changes: `+1 + -1` which is, of course, zero. Since there is no change, the sum of i and j is invariant.

In this case, we could dispense with the `j` variable provided we don't need it outside the loop, and replace it with a calculation based on the inveriant:

```
    for(int i=0; i<100; ++i)
      cout << 100 - i << "," << i << endl;
```

This requires a *proof of correctness,* which is seems fairly trivial: since $j+i = 100$ then subtracting i from both sides gives us that $j = 100-i$. So the combination of an invariant, or law, pertaining to the original loop body, together with some mathematics, yields an optimisation which can be proven correct.

1.1.2 Incremental Programming

Optimising code believed to be correct by applying successive small changes which can be proven to preserve the semantics, is a powerful technique I call *incremental programming.* It allows code to be written which is first proven correct, but may have poor performance, and then improving the performance whilst continuing to maintain correctness.

This technique is very valuable because the original reasoning establishes correctness, and then small easily proven steps are used for optimisation, where the steps are basically proven easily by showing invariants are maintained, or, as in the example above by using the invariant as the basis of the transformation.

A related technique involves changing the semantics. Again, we change a small part of the code, assuming the rest of program semantics are invariant. Then if something goes wrong, it is most likely in the code we just changed.

1.1.3 Representation Independence

Consider:

```
vector<int> x = ...
for(int v : x) {
  // complicated calculations
}
```

We don't like using `vector` so we change it to `list`.

```
list<int> x = ...
for(int v : x) {
  // complicated calculations
}
```

We reason this is safe, and preserves correctness, because the iteration still scans all the values of the data structures in the same order, and the calculation only sees the values. We may need to change how we put things into the data structures, but if we get the same order, the complicated calculations should still produce the same result.

This is an example where the loop and calculations are *representation independent,* which is a kind of invariance. In this case the C++ language is providing direct support for this idea with templates and iterators.

1.1.4 Representation Invariants

Another popular example of the utility of the concept of an invariant is shown here:

```
class rational {
  int numerator;
  int denominator;
public:
   rational(int num, int den) {
     if(den == 0) throw "zero denominator";
```

```
        int sign = sgn(num) * sgn(den);
        unsigned int uden = abs(den);
        unsigned int unum = abs(num);
        unsigned int g = gcd(uden, unum);
        numerator = sign * unum / g;
        denominator = uden / g;
      }
      ...
  };
```

Here the constructor is dynamically enforcing an invariant that the denominator of the representation is a positive integer which is relatively prime with respect to the numberator. This ensures the denominator cannot be zero, and if the rational number is negative the sign will be found in the numerator.

The division by the greatest common divisor of the the input values ensures the stored representation is minimal, that is, the values are as close to zero as possible. This increases the set of rational numbers available to the maximium possible with the given representation.

Now we will write a method to multiple the value by another rational number:

```
    operator *= (rational other) {
      int num = numerator * other.numerator;
      int den = denominator * other.denomintaor;
      int g = gcd(num,den);
      numerator = num / g;
      denominator = den / g;
    }
```

First we need to check that the representation invariant is maintained. Clearly we have code there to ensure the relatively prime part is obeyed. But now we see the advantage of an invariant in two ways: we obtain both optimisation of the code *and* ease of reasoning as a result: the new denominator must be positive because the product of two positive values is positive, and because we assume the gcd of two integers is always positive.

The reasoning is easy and the code is shorter than in the constructor so not only is our multiply likely to be correct, the resulting representation maintains the invariant.

It may seem this is a perfect piece of code but alas it is a not, it is a very bad piece of code as we shall see in the next section!

Here is why our code is bad:

```
int main(){
   rational x(5,7);
   void mul(thread.id *pid, int n, int d) {
     pid = thread.get_id();
     x *= rational(n,d);
   }
   thread.id id1;
   thread.id id2;
   thread (mul, &id1, 42, 7);
   thread (mul, &id2, 16, 8);
   thread.join(id1);
   thread.join(id2);
   cout << x << endl;
}
```

We spawn two threads, each of which multiplies x by a rational number, join the threads with the main thread and print the result.

Unfortunately, the multiplication method is not thread safe, and there can be several *race* between the two multiplications. First, both threads could fetch the old value at the same time, do the multiplies in some order and store the results in order, but then the result would be the orginal value muliplied by one or other of the new values when we wanted it to be multiplied by both.

But worse is possible! When storing the new representation the first thread might store the numberator, then the second stores the numerator and denominator then the first finally stores the denominator. The result is not only wrong, there's no assurance the relatively prime invariant is maintained.

We can fix this by adding a lock to the representation and locking it at the start of the multiplication method, releasing it when finished. But then we are paying a price: we're using extra space and incurring a performance hit, even if our application is single threaded.

Instead, let us try to use inheritance to solve the problem:

```
class rational { ..
protected:
  void unsafe_mulby(rational other) {
    int num = numerator * other.numerator;
    int den = denominator * other.denomintaor;
    int g = gcd(num,den);
    numerator = num / g;
    denominator = den / g;
  }
private:
```

```
    virtual void mulby(rational other) { unsafe_mulby(other); }
  public:
    void operator *= (rational other) { mulby(other); }
  };
```

Note the serious language design problem here: we had to invade and rewrite the original class so it is a suitable base class for the derived class:

```
  class ts_rational : public rational {
    atomic<bool> lock;
  public:
    ts_rational(int n, int d) : rational (n,d), lock(false) {}
  private:
    override void mulby (rational other) {
      while(lock.exchange(true, memory_order_acquire)); // lock
      unsafe_mulby(other);
      lock.store(false, memory_order_release); // unlock
    }
  };
```

This code is a lot more complicated! The core multiplication is put in the `unsafe_mulby` routine. This method is protected, so it cannot be called by the public, but it can be called in a derived class.

We then define a virtual method `mulby` which is entirely private, so it can only be called in the `rational` class, and then we define the public `operator *=` method to call it.

Finally in the derived class, we need only define the overriding `mulby` method which calls the `unsafe_mulby` method to do the work inside a critical section provided by the spinlock.

The structure presented here is the *only* correct way to organise this. Note that contrary to popular myths propagated by some very well known authors, virtual functions in C++ should always be private!

Furthermore, the basic public method of the base class should generally be wrappers dispatching to these virtuals. In pathicular overriding virtuals should never be called explicitly, and should actually be entirely hidden, only C++ provides no hidden access modifier: another design problem in C++. The only way the override should ever be invoked is by a virtual dispatch from the call in the base.

Our code indicates an important principle: public constructors must establish public invariants, and public methods must maintain them. However, public methods should only be called by the public! Never call public methods from any methods in your class!

The reason for this rule is simple enough: public methods must do extra work establishing and maintaining representation invariants. In addition, the body of a public method may often be augmented with debugging code to trace the history of public access to the class, and we don't went internal calls to mess up the tracing.

In fact, non-public methods need not maintain public invariants: public invariants are pre- and post-conditions on public methods. Once the pre-condition is established an inner call in a public method can assume the pre-condition rather than checking it. The post condition of an inner call need not obey any rules at all, because more work may be done.

So, all in all, we appear to have solved the problem conditionally: we have imposed a responsibility on the programmer to use the `rational` class if the application is single threaded, or use the `ts_rational` class if multi-threaded.

Also the code is more complex and harder to reason about.

Don't believe me? Well you'd better because the code is *still wrong!*

The problem is simple enough: there is an operation we are using implicitly which is provided by the compiler and the compiler implementation is not thread safe. Note we pass the argument to the multiplication method by value which might invoke the default copy constructor. This will copy the non-static data members `numerator` and `denominator` one after the other with a pre-emption possible in between so the values might be changed by another thread half way through. We have another data race!

How can we fix this?

Object Orientation is Rubbish

You are going to hate the answer. It cannot be fixed! If you are a fan of object orientation it is about time you woke up. Object Orientation is a false paradigm. It must be abandoned because quite simply it does not work. This is not to say that classes and virtual dispatch are not useful! They certainly are, but they do not solve all problems.

In particular any problem in which functions have two variant arguments cannot have an OO solution. This has been known for decades and is the problem is called the *covariance problem.*

Let me elaborate the issue with our example. The default copy constructor for `rational` is perfectly good in a single threaded context. However to maintain the representation invariants in a multithreaded context, not to mention actually getting the right abstract value, it must do the copying atomically.

To do this a lock is required and we have one! The problem is it is in the wrong place! It is in the object, not the argument, and the argument type `rational` makes no provision for a lock!

So we can fix that with a copy constructor for `ts_rational`, right? Certainly we can do that:

```
ts_rational(ts_rational const& other) {
  while(lock.exchange(true, memory_order_acquire)); // lock
  numerator = other.numerator;
  denonminator = other.denominator;
  lock.store(false, memory_order_release); // unlock
}
```

The problem is, this constructor is not called passing a `rational` argument, but a `ts_rational` argument so lets change the `mulby` argument types to `ts_rational`, ok?

Woops! Its not ok at all because now `ts_rational::mulby` no longer overrides the base class virtual! And we can't change the base class virtual method either because that would break encapsulation and a lot of other basic principles.

What's the solution? The answer is simple: abandon object orientation altogether and lets try another paradigm.

1.2 Functional Programming

Functional programming is ideal for performing calculations. One of the most basic principles is that objects are *immutable.* Combined with the idea that functions should be *pure*, this provides a powerful property known as *referential transparency.*

Let us recode our example using functional programming concepts:

```
rational operator * (rational other) const {
  int num = numerator * other.numerator;
  int den = denominator * other.denomintaor;
  int g = gcd(num,den);
  int n = num / g;
  int d = den / g;
  return rational(n,d);
}
```

Notice carefully this method multiplies the object by the other argument and returns a brand new value.] This code is not yet good because a data race still might be possible. Let us assume the immutable property for the moment, noting that the `const` qualifier on the method ensures that the object cannot be modified. The problem we have here is that the `rational` constructor is invoked and although it does the right thing, it breaks the rule that public

methods should never be called by any method. Consequently it does a useless check for zero, fiddles the sign calculation and find the gcd and divides by it pointlessly, because it is already done.

To fix this, we need another constructor:

```
  private:
    rational(int n, int d, int dummy) : numerator(n), denominator(d) {}
```

We added a useless dummy argument because C++ doesn't provide another way to do this. In fact we are going to recode the public constructor to use it as well:

```
  class rational {
    int const numerator;
    int const denominator;
  public:
     rational(int num, int den) {
       if(den == 0) throw "zero denominator";
       int sign = sgn(num) * sgn(den);
       unsigned int uden = abs(den);
       unsigned int unum = abs(num);
       unsigned int g = gcd(uden, unum);
       new(this) rational(
         sign * unum / g,
         denominator = uden / g,
         0
       );
     }
     ...
  };
```

This is a dirty technique! Languages have limitations and C++ many. Luckily there is a workaround! The problem is we want to make the non-static members `const` to be sure they cannot be modified. Unfortunately this prevents assignment. But there is a loophole in the C++ type system, that inside a constructor the `this` pointer is always non-const, precisely to allow assignments in the body. In this case we still can't do assignments, but we can use a placement new calling our private constructor to do the job.

It is not necessary to make the data members const, provided all the methods do not modify these values: this can be assured by making the methods const.

However making the data const is a great aid to reasoning because it localises the assurance to two lines right at the top of the class, rather than having to troll through all the methods checking they're all const.

Localistion is a powerful tool used to make reasoning simpler.

Now you might say: well so what?

Well, our class is now thread safe!

Hey, what? Why? Well its clear, even if there is a race to get at the two data members when copying a value or otherwise, it doesn't matter because the variables cannot be changed! The only thing that matters is that the values are both set before we access them, and this is assured in normal good code because the constructor is always executed in a single thread and completes before a binding to the result is establshed. Of course, if you use say a placement new to initialise a variable two threads have access to you *already* have a problem that the threads could look at the variable before initialisation even starts. The point is, the functional code cannot introduce any new problems!

1.3 Optimisation

Optimisation is a process whereby a human programmer of compiler is able to improve the performance of a program by modifications to the code which do not change the semantics.

One of the most common optimisations is to perform experiments on a program which has several parameters which mediate tradeoffs so find which combinations lead to the best performance for particular kinds of data sets. This is known as *parametric tuning*. It is very common to tune garbage collectors.

There are specialised tools to help identify performance issues called *profilers*. They are generally easy to deply to check for bottlenecks but quite difficult to use for tuning.

1.4 Localisation

Localisation is an important technique to improve the ability to reason about code. For example consider the following

```
struct X {
  int y;
  void inc();
};
void X::inc(){ ++y; }
```

compared to

```
struct X {
  int y;
```

```
  void inc(){ ++y; }
};
```

The second case is shorter and the encoding of the `inc` method involves incrementing the non-static member `y` which is defined on the line above. In the first case we have to look three lines up to find `y` is an `int` and two lines down from the declaration of `inc` to find its definition. Worse, if the signature of `inc` changes we also have to change the definition. In addition, the name of the struct, `X` has to be repeated and so when seeing the out of line definition we have to again look up to find the struct declaration.

Another example:

```
mutex lock;
void f() {
  lock_guard<mutex> locked(lock);
  // do stuff holding mutex
}
```

illustrates the Resource Acquisition Is Initialisation (RAII) idea in C++. Using the guard not only acquires the lock, it ensures the lock is held for the whole scope of the function `f`, and what's more, it is correctly released when the function exits, even if it does so as the result of an exception being thrown. In this case a single line not only acquires the lock, but also ensures its release. Compare this with:

```
mutex lock;
void f() {
  try {
    lock.lock();
    // do stuff holding mutex
    lock.unlock();
  }
  catch (...){
    lock.unlock(); throw;
  }
}
```

I hope this is equivalent but it could fail is the locking the lock throws an exception! Can this happen? I have to look up the C++ Standard to find out. TL;DR. The code is unreliable. It is longer, it is clearly harder to decide if the code is correct, and the code is fragile because we have to be careful modifying it. What if we needed to lock two locks? What happens if the unlock operation fails? Can that throw an exception? Does it matter?

The utility of localisation is not merely a coding technique for it depends on the language design supporting it. In C++ the coupling of object destruction to release associated resources with the release of the object memory in the the `delete` expression and the ability to define for a class the destructor performing the relevant operations inside the class and lexically close to constructor and data members holding the resources is a significant aid to reasoning. It is certainly better than writing the release code every time an object is freed!

Such is the strength of belief in localisation of this kind that many languages have been designed to support this kind of localisation, and in other cases, new versions of the language added support later.

Here's an example from Java:

```
class TestFinallyBlock {
  public static void main(String args[]){
    try{    }
    catch(NullPointerException e){  }
    finally {  }
  }
};
```

The `finally` clause is part of the `try/catch/finally` statement, and provides a local assurance some code will be executed whether or not an exception is thrown, in other words *always*. Knowing something always happens is easier to understand than something that happens depending on a condition.

The same idea in golang:

```
package main
import "fmt"
func main() {
        defer fmt.Println("world")
        fmt.Println("hello")
}
```

The deferred operation is invoked at the end of its containing scope, but it written at the start.

A related notion is that of a lexical closure. In Ocaml for example:

```
let f x = let g y = x + y in g
```

the function `g` is returned from `f`, closed over its environment, which contains the parameter `x` of f, as well as its own parameter `y`. The point of a lexical closure is that by specification it implies proximity of physical code, that is, the

actual characters of text: that's what lexical means. So the idea of a lexical closure is you can reason about what the function does only looking in a small space in the text file.

In summary, localisation is a way of enabling *local reasoning*. We need to complete our survey with a counter example! Hindley Milner type inference is our example .. and it explains why I hate it! From Ocaml:

```
let f g = g 1 in
let h g = g 42.3 in
let k g1 g2 x = g1 x + g2 x in
let g i = k f h i in
g 1
```

This gives an error but it is not clear unless you duplicate the compilers reasoning. The function `f` accepts a function which accepts an integer, and `h` accepts a function which accepts a float. The function k takes two functions which accept the same type and return an int. Now, when k is applied inside g it passes f and h which apply each to the same value parameter x must be both a float and an int which is impossible so we get an error in function `k`. But actually we meant to write:

```
let f g = g 1.0 in
```

as the first line. So the error message is entirely wrong. the problem is the Ocaml compiler isn't smart enough to track how it made the inferences and report all the possible sources of the error, which is a good thing also since the inference is accumulated by refining the set of possible types on ever line of the program above.

A proper error report would resemble the mess you get with template instantiation errors in C++: the compiler simply doesn't have any way to determine where the real error is because it is using global reasoning instead of local reasoning.

When Microsoft introduced type inference into C# they took great care to ensure it only worked locally, precisely to prevent the above problem.

It is instructive, in the Ocaml example, to learn how to find the error: one adds type annotations for example:

```
let f (g: int \rightarrow  int) = g 1 in
...
```

and now we have localised the inference by forcing it to agree with the parameter immediately, which immediately finds the real error. The point again is that

lexical proximity can provide local reasoning.

There are many programmers like me, who do not actually read code! It's true, if you give me code to examine I won't read it. I just look at it wholistically. If it isn't indented properly I'll throw it back at you or immediately edit it so it is. I can't read code that doesn't have structure because I learn what the code does from its visual structure.

Humans eyes do massively parallel processing and can see two dimensional patterns almost instantaneously. Reading code is a linear exercise which requires scanning laboriously and remembering what you've already read. The best way to understand code is to only read the bits you actually need to, and the wholistic pattern matching is how to identify which bits need further examination and thought.

Unsurprisingly compilers are better at local reasoning too! In particular complex analyses such as data flow analysis are almost universally done locally, within a single function only, and then, the local results painfully coupled with some ad hoc algorithms, since the analysis is an intractible $O(N^3)$ or worse. This is fine inside a small function, but we need to find a way to combine the results which is $O(NlogN)$. This is the biggest number which is considered scalable.

1.5 Anonymity

It is said a witch is undone if here secret name is revealed. So too with programming, anonymity is power. Consider

```
auto x = 1 + g (f (a, b), c);
```

There are two unknown functions and three unknown variables here, which is five non-local names we have to lookup so we can understand the computation. Also we are not sure what the resulting type is, but it must be something an `int` can be added to.

That's a lot of work but this is worse:

```
auto tmp1 = f (a, b);
auto tmp2 = g (tmp1, c);
auto x = 1 + tmp2;
```

The lack of type information on the temporary variables is not new. But that's two additional variables we have to deal with. It may seem since their definition is local to their use, that this is not too bad, but in C++ we do not know how often or where a variable is used until the end of the scope it is declared in. The compiler may have difficulty too.

A variable which is only used once as a function argument can be treated specially: it can be elided completely. This elision is apparent lexically by examining the first case again, where there are no variables named `tmp1` or `tmp2`. However the elision of the name is not all that happens: the store itself can be elided! In fact this is mandatory in C++20.

Instead, the result of computing `f(a,b)` will initialise the first parameter of the function `g` directly without first constructing a temporary, copying it to the parameter, and then later deleting it.

It helps to understand the basic stack machine model. Here is what happens when you call a function:

1. Space is reserved on the machine stack for the result
2. the return address is pushed onto the stack
3. the arguments are pushed onto the stack
4. the program counter is set to the function entry point

Now to return a result here is what happens:

1. the result is stored in the reserved slot on the stack
2. the program counter is set to the return address
3. the stack is reset to point at the result concurrently

The program now continues where it left off with the result on the top of the stack. Now if you consider the call of the function `g` above in the first case, when it gets to pushing the first argument, namely `f(a,b)` is recursively performs the call sequence, which invokes f and at the end of that leaves the result sitting on the stack exactly where `g` needs its argument to be. In fact, that slot on the stack *is* the first parameter of `g`: parameters of functions and fixed offsets from the stack pointer .. as is the result!

The critical point is that, without optimisation in the second case, the variable `tmp1` has to be copied onto the stack, and worse, the result of the computation of `f` has to be copied into that variable as well. So two extra copies would be done!

1.6 Indeterminacy

Many programmers think indeterminacy is a bad thing, as do many language designers. This is a serious problem because the indeterminacy is a key way to simplify code. Simple code is likely to be faster, use less resources, and be easier to reason about.

Lets look at some examples and their consequences. In many languages the order of evaluation of function arguments is unspecified. In languages with eager

application evaluation semantics like C, the arguments must all be evaluated before the function is called. Notice that evaluation of arguments is not entirely undefined! The order is unspecified but there must be some ordering which is bounded by something determinate: completion prior to calling the function. This is the Yin and Yang of indeterminacy: it is always bounded by something determinate.

What are the consequences of the indeterminacy? Obviously the compiler can now choose any order. It may choose an order so it can push the arguments conveniently onto the stack. Or it may, if it can be sure of referential transparency, lift out common sub-expressions, or even interleave the computations.

It is worth mentioning interleaving because it is a very powerful optimisation on modern processors. If you have two expressions to evaluate there will be some kind of tree of dependencies of inner function calls and their arguments which are also expressions. If you take all the arguments of the function call and unravel them into three address code into a sequence of assignments of simple operations, then you can see that the sequence can be reordered many ways to get the same result. Consider:

```
f (x + y * z, a + b * c)
```

admits the ordering:

```
t1 = y * z
t2 = x + t1
r1 = b * c
r2 = a + r1
f (t2,r2)
```

But this is a *bad* ordering because the evaluation of t2 must stop and wait for t1 to be evaluated, holding up the evaluations of r1 and r2. This is a much better ordering:

```
t1 = y * z
r1 = b * c
t2 = x + t1
r2 = a + r1
f (t2,r2)
```

because the first two computations do not have any interdependencies and so can be done simultaneously. Modern CPUs have high speed pre-loaded instruction pipelines and multiple ALUs and can certainly do this kind of thing.

Now, in a programming language like C++, expressions can have side effects. These effects may be coupled and depend on ordering. The compiler can always

try to see if a deterministic ordering specification can be changed in a way the programmer cannot observe (in C++ this is called the *as if* rule). But doing so can be hard.

But it is not at all hard if the language *specifies* the ordering is indeterminate! In this case if the behaviour has changed, the programmer is responsible for incorrectly depending on something which was not assured. The compiler is absolved of all blame and is able to perform optimisations much more easily!

1.7 Abstraction

1.8 Reentrancy

Reentrancy is a property of code that allows the code to be entered by any thread at any time so that elaborations of the code do not interfere.

If a function is fully reentrant it is *automatically* capable of recursion and execution by any number of threads simultaneously.

if a function is pure, depends only on value parameters, and uses only local storage, it will be re-entrant. Most implementations of mathematical functions like `sin`, `cos`, `sqrt` are reentrant. Most reentrant functions can also be used in signal handlers because reentrancy guarrantees non-interference with other code, however, in some OS and with some hardware, interrupts do not save and restore the whole machine state. For example, expensive registers like floating point registers might not be saved to reduce interrupt service latency, and so aren't available for use at least unless saved by the interrupt service routine.

Posix specifications will list which library functions can be used in signal handlers.

1.9 Boundary Conditions

For any algorithm there are found conditions:

1. the initial state used to get the algo rolling
2. the usual processing step to progress the algorithm
3. the terminal state when we have the solution
4. special cases

The hallmark of good design is then the boundary conditions, the initial and terminal states, are handled automatically by the usual processing step, and there are no special cases.

A sign of poor design is the repetition of a test already done.

It is not always so easy to see how to do this: design is hard! Consider the following almost trivial problem: you have two tapes with numbers written on them in ascending order, and a special value larger than any at the end which acts as an endmarker. You need to write the numbers out on a third tape in ascending order: this is a classic algorithm called merge sort.

The body of the algorithm is clear:

```
if(v1 > v2) { write (3, v2); read(1, v2); }
else { write (3, v1); read (2, v1); }
```

Now the initial setup is clear as well:

```
read(1, v1); read(2, v2);
```

The variables v1, v2 are buffers and this step is known as priming the buffers.

The terminal condition, however, is not handled correctly: it occurs when both variables contain the end marker. One way to do this efficiently is as follows:

```
  read(1, v1);
  read(2, v2);

next:
  if(v1 == endmarker) goto finish2;
  if(v2 == endmarker) goto finish1;

  if(v1 > v2) { write (3, v2); read(2, v2); }
  else { write (3, v1); read (1, v1); }
  goto next;

finish1:
  write(3, v1); read(1,v1);
  if(v1 == endmarker) goto finish;
  goto finish1;

finish2:
  write(3, v2); read(2,v2);
  if(v2 == endmarker) goto finish;
  goto finish2;

finish:
  write(v3, endmarker);
  return;
```

But this is not very nice! if on value if greater than the other we're writing out the smaller value, even if the larger one is the endmarker! So the normal processing suffices when at least one tape is still running:

```
next:
  if(v1 > v2) { write (3, v2); read(2, v2); }
  else if (v2 > v1) { write (3, v1); read (1, v1); }
  else if (v1 == endmarker) goto finish;
  write (3, v1); read(1, v1);
  goto next;
```

This is better because it only checks for an end marker wuen the two variables are equal. But there is still a code smell there, since the code to write v1 out is repeated. The trick is to notice we have to write the endmarker anyhow and when we do, we're done:

```
next:
  if(v1 > v2) {
    write (3, v2);
    read(2, v2);
  }
  else {
    write (3, v1);
    if (v1 == endmarker) return;
    read(1, v1);
  }
  goto next;
```

This seems optimal! The variable `v1` is examined twice and `v2` only once.

But is it? In the old days data was entered by keypunch operators onto punch cards during the day. Then the night operator grabbed all the cards and put them in a card sorting machine. Finally the cards are sorted and put into the card reader, the master input tape is mounted, and a fresh output tape is mounted. Now the merge sort is run.

The master tape contains a full year of data, and the output tape contains the data updated with the days new input. The point is the master tape is big, but the input is relatively small. When the cards are all read, the rest of the input tape could be fast copied to the output, saving time. The original algorithm does this: once one input is exhausted the other can be copied without any comparison except for a check for the end of the tape.

In addition, althougth considerably longer the original algorithms is "obviously" correct whereas the final algorithm requires some thought to verify its correctness. In particular the termination condition that `v1` is the endmarker requires

an additional non-local clue to verify: the previous condition that `v1 > v2` when reversed yields the condition `v1 <= v2` which when combined with the fact that the endmarker is larger than all values other than itself, implies that `v2` must contain the endmarker also. For, were this not the case, it would have to be less than the endmarker, and therefore less than `v1` contrary to the reversed condition which says it is greater or at least equal!

We have proven the correctness of the termination condition, but the point is we had to work hard to do so. It is not obvious at the first glance! How can we chose the best algorithm?

It is not clear! It is a style issue, perhaps we may side for the longer more obvious algorithm based on the *KISS* principle: Keep It Simple, Stupid! But perhaps we prefer the more compact solution. In the latter case it would be a good idea to add comments suggesting the proof of correctness, which would make the algorithm longer!

1.10 Bounds

Computers are finite state machines and this means everything is bounded. Consider for example our representation of rational numbers. Since the numerator is an `int` it is bounded above by `INT_MAX` and bounded below by `INT_MIN`, inclusive. Clearly, the rational numbers represented are also bounded above and below by the same values.

When you write a loop like:

```
    for(int i= 0; i < 20; ++i) ...
```

then variable `i` is called the *loop variant* because it changes, the initial value of `i` is 0, and the final value is 20, and these are bounds on the variant. Note that the loop body executes from the lower bound up to but not including the upper bound.

In many algorithms, we have a bound on the number of iterations, which the algorithm may not actually reach. For example consider the computation of the square root of a large integer K. The method is to guess at $Q = K/2$ and see if it is too big or too small or spot on by squaring it.

If it is too small, add one and try again, until it is too large. Now we know the answer is between our current guess and the previous one.

Similarly, if it is to large subtract one and try again, until it is too small, then the answer is between the current guess and the previous one.

In both cases we do not know the exact answer but we have a lower and upper bound for it.

Another question to ask is: how many iterations do we do? We do not know the exact number. But we must do at least two, to find the upper then lower bound, or lower then upper, unless we hit the exact answer immediately. On the other hand, we cannot do more than $K/2 + 1$ iterations because the lower bound on the square root of a positive integer has to be at least 1, and cannot be more than the number itself, and since we started in the middle, we'd run out of options after that.

So we also have bounds on the number of iterations.

This method may be bad, and the bounds may be weak estimates, but the point is, we have bounds. A bound is not an average value it is a hard limit. It is often the case an algorithm is too complex, or the inputs may vary in some range, so we cannot know exactly how long it will take, or how much storage it will use. But we should always be able to calculate an upper bound, otherwise we cannot know when a bug has forced it into an infinite loop.

By far the most important general case of this which is regularly ignored are bounds on recursion. All recursions should be bounded! If the bound cannot be proven, a run time test should be used to report the expected bound is exceeded.

For example consider a parser processing a program which recurses into nested blocks. It is reasonable to set a bound of 50 on the recursion as a way of trapping a syntax error: no sane programmer would write code with 50 levels of nested blocks!

Chapter 2

Math Review

This is the bit many feared would come and here it is! The maths we need for programming generally, however is not linear algebra, nor calculus .. but really quite basic algebra.

However, we will be introducing something most programmers need to know but probably don't: *category theory.* But first ..

2.1 Set theory

Most programmers think they know the basics of set theory: sets, intersections, unions, maps. Actually most fail to understand the core idea of set theory as a way of dealing with equality.

Equivalence Relation

Definition 2. A relation $A \sim B$ is an *equivalence relation* if it is *reflexive* $A \sim A$, *symmetric* $A \sim B$ implies $B \sim A$ and *transitive* $A \sim B$ and $B \sim C$ implies $A \sim C$.

Lemma 1. *Let S be any set with an equivalence relation defined on it, then the set can be* partitioned *into subsets called* classes *which are pairwise disjoint, and the union of which is the whole set S.A*

Now if someone asks you what equality is the complete answer is *any equivalence relation.* It's somewhat surprising but there is no deeper possible definition.

Now, you will guess correctly, this means there can be more than one notion of equality and this is correct. Indeed, this is a key factor in the notion of abstraction!

Definition 3. Suppose we have two equivalence relations on the same set, $A \sim B$ and $A = B$ then there is an equivalence relation $A \equiv B$ defined to hold if, and only if, both $A \sim B$ and $A = B$. This is the *product* of the two relations.

It is immediately evident the corresponding partition consists of all the intersections of the classes of both partitions, with the property that any class is a subset of a class of both argument partitions. Such a relation or partition is a called a *refinement* of each of the argument partitions. Importantly no class of the refinement overlaps a class of the input, each class is equal to or wholly contained.

What this means is that for any two definitions of equality there is always a refinement which satisfies both.

Maps

In set theory a map, sometimes called a function, is a special kind of binary relation with the property that each value of the left hand set, called the *domain,* is associated with exactly one element of the right hand set, called the *codomain.*

Its very important to note that the domain is a single set. Mathematics has no support at all for functions with multiple arguments. Very large numbers of programming languages get this wrong and it always leads to serious design faults in the language.

There are three special kinds of maps you need to know.

Definition 4. An *injection* is a map that carries every element of the domain to unique element of the codomain. It is sometimes called a 1 to 1 map.

Definition 5. A *surjection* is a map that targets every element of the codomain. Thus for each element of the codomain there is at least one element of the domain which maps to it.

Definition 6. An *bijection* is both an injection and surjection and is sometimes called a 1 to 1 correspondence. It has the special property that there is an *inverse* map going in the other direction that "undoes" the original map.

2.1.1 Representations

The math above is very important. Lets consider our rational numbers again, with the form (p, q) representing the fraction p/q where q is not zero, but as yet the other rules we used before are not there.

Our objective to that this pair of integers be a representation of the rational numbers. This means we have to write operations like multiply and add, that work the same way on the representation as the abstraction. To be specific consider the representation map $R : int * int \to Q$ then say, addition on the representation must obey the rule:

$$R(a) + R(n) = R(a + b)$$

In other words, if you add the rational numbers associated with the representations a and b together, you should get the same rational number you would get but adding the representation using the add method you coded, and then figuring out which rational number the result represents.

We will see later how to express this in a high level way. For now I want to consider that $(1, 2)$ and $(2, 3)$ represemnt the same rational number. The representations are not equal, but they are equivalent in some sense.

More generally given any map $f : A \to B$ at all, we can define an equivalence relation on A by the rule $a\ b$ if and only if $f(a) = f(b)$. Immediately, we have a partition on A.

Now, we can define an *injection* from the partition on A to B, by saying each class in A maps to the element of B any value in the class does: they all map to the same element because that was the very definition of the equivalence relation.

It's common to denote the class that an element a is in with the notation $[a]$. Now sometimes we want to give a specific name to the class, and it is common practice to systematically pick a particular element of each class as the name: this element is called the *canonical representative* of the class.

For rationals lets pick the canonical representative as the value of p and q where q is positive and p and q are relatively prime. We can do that because those integers are unique within the class of equivalent pairs representiung the same rational.

Hey! That's the invariant we chose before!

So now you know what a representation invariant is and why it's useful. More precisely, a representation invariant is the logical condition describing a subset of the set of all possible values of a representation. It acts as a pre- and post- condition on public mutators, a post condition of constructors, and a pre-condition of destructors.

Restricting valid state of a representation to canonical values is generally good practice because it reduces the complexity of computations, improves performance, makes reasoning easier, and can extend the range of abstract values represented.

Chosing representations and their invariants is a design challenge.

Chapter 3

Category Theory

It is now time to meet the *universal algebra* which is the *theory of abstraction.* It is called category theory.

3.1 The category Set

Consider a collection of sets, and all the functions between them. We will have an operator on the function, namely function composition. Because I said all the functions are included in the collection, the composite of any two functions, it it exists, must also be in the collection. For a composite $h : A \to D$ of the functions $f : A \to B$ and $g : C \to D$ to exist, defined by $h(a) = g(f(a))$ it is necessary that $B = C$. The composite in the usual notation is written

$$h = g \circ f$$

which is the forward notation. However we much prefer reverse composition

$$h = f \odot g$$

in which it is more obvious f is applied first, then g.

You should observe composition is associative, so that because

$$(f \odot g) \odot k = f \odot (g \odot k)$$

we can omit the parentheses and just write

$$f \odot g \odot k$$

Since the all the functions are there, for every set X there will be the identity function 1_X which maps each element of X to itself.

What we have described is an example of a traditional category. More generally, categories are any collection of *objects* and *arrows* with an associative binary composition operator, identities, and such that the set of arrows are closed under composition; that is, every composite is actually in the collection.

3.2 The Category Monotype

The main computer science category we will be considering is closely related: it is the collection of *types* and *functions.*

The key idea of category theory is that we work *only* with the arrows. The collection of objects are in a one to one correspondence with identity arrows, and their only purpose is to tell which arrows join up, that is, can be composed.

It is vital to understand this concept. There is a related construction called an *arrows only category* which has no objects at all, but the construction is more technical so we usually allow the objects with the caveat that they are merely a convenience and have no properties of interest.

Lets translate that into the computer science example: in a categorical interpretation of computing, types have are of no interest!

What, you say? You thought types were important. That integers were interesting. No! Integers have no structure. In the abstract viewpoint *only* the functions operating on integers .. and on other types .. have any significance.

This is a radical viewpoint for most programmers! There are no values of type, because types are not sets of values, they're merely a way of telling which functions can be composed. It is the functions which have structure!

I am going to show you that category theory subsumes set theory, and that it is a powerful new way of thinking. In the tradition of Einstein, we have to show two things with thought experiements: that we do not in fact lose anything we already know with this new perspective, but rather, the paradigm shift allows us to generalise that knowlege to new cases.

3.3 Isomorphisms

Let us begin with an easy example. The examples will get harder for a while until the paradigm shift kicks in. Lets start with the idea of a bijection. Given two sets A and B a bijection is a function $f : A \to B$ such that each element a of A maps to a *unique* element $b = f(a)$ of B, and, in addition, for every element b of B there is an a in A that f maps to b. Therefore there is a one to one correspondence between the elements of A and B.

The uniqueness condition assures us that if $f(a1) = f(a2)$ then we must have that $a1 = a2$, the onto condition, that every element of B is mapped to, means for every element b we can pick out at least one element A that is mapped to that b, and together, these conditions assure us that the a involved is unique.

In other words, there is a function, written f^{-1} which goes back in the other direction, and is also a bijection. Furthermore, there is only one such function, for the rules clearly determine it.

So now we can restate that by saying that if f is a bijection then it is invertible.

Wait there! Read that again! Did I mention any set elements in that property? No! Invertibility is a property of the function which does not mention any values of the sets!

In category theory, an arrow which is invertible has a special name: it is called an *isomorphism.* Iso in Latin just means equal or equivalent.

But we haven't defined what invertible means in terms of arrows! Lets do that now: the functions $f : A \to B$ and $g : B \to A$ are said to be inverses if and only if:

$$f \odot g = 1_A \text{ and } g \odot f = 1_B$$

This just says the composite of f followed by g gets you back where you started, because it is the identity function for A, and the composite g followed by f is the identity function for B because it gets you back where you started.

It is a fact you can prove that, if f is invertible, its inverse is unique. There is only one inverse. This allows use to write it as f^{-1}.

Now the point here is that the set theoretic notion of an bijection, based on how a function maps set elements, is fully described in categorical terms, that is, in the abstract, by simply saying the function is invertible: every invertible function in the category **Set** is a bijection, and every bijection is invertible.

We have dispensed with all that talk about set elements and given an *abstract* specification, meaning, one characterised entirely in terms of functions and composition in the category.

Now we can generalise! An arrow is called an isomorphism, in *any* category, if it is invertible.

3.4 Monomorphism

Now let us consider the notion of an injection. This is a one to one map $f : A \to B$ with the property that if f maps two values $a1$ and $a2$ to the same element b in B, then $a1$ and $a2$ must be equal, that is, $f(a1) = f(a2)$ implies $a1 = a2$.

The corresponding categorical notion is that of a *monomorphism*, and we can say that an arrow is *monic*. How can we charcterise this in the abstract, as a property of the arrows alone, without mentioning set elements?

Suppose we have two maps $g1 : X \to A$ and $g2 : X \to A$ and suppose $g1$ takes x to $a1$ and then f takes it to b. Now suppose $g2$ takes x to $a2$ and then f takes it to b again. It follows that $a1 = a2$ because, since f is an injection, distinct elements go to distinct elements, and so since its the same b, the two a's must be the same too.

So in terms of maps, this is saying that if $g1 \odot f = g2 \odot f$ then it follows $g1 = g2$. Hey, we said that without mentioning any set elements! It is an abstract definition, we will take this as the definition of a monomorphism.

3.5 Epimorphism

Finally consider a surjection. This is a map $f : A \to B$ that covers all of B, that is, for any element b in B there is an element a in A such that $f(a) = b$.

Suppose we have two maps $h1 : B \to C$ and $h2 :\to C$ and suppose $f \odot h1 = f \odot h2$ then it follows by considering elements that $h1 = h2$. We take this abstract requirement in terms of functions as the definition of an *epimomophism*, and also say f is *epic* as this is a quality item!

3.6 Application to types

In the category **Set** if A is a subset of B, then there is a monomorphism from A to B which takes each element to itself. More generally, with renaming of elements, any injection defines its domain as a kind of subset of the codomain.

In the category **MonoType**, if there is a monomorphism from A to B we can consider A to be a subtype of B. In practice, these monomorphisms may actually exist as functions which are called *coercions*.

3.7 Products

We are familiar with the binary cartesian product of two sets, A and B which is notated $A \times B$ and defined as the set of all pairs (a, b) where a is in A and b in B.

In category theory a product is a pair of morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ called *projections* satisfying the following rule: for every pair of functions $f1 : C \to A$ and $f2 : C \to B$ there is a unique function $f : C \to A \times B$ called the *mediating morphism* such that $f1 = f \odot \pi_1$ and $f2 = f \odot \pi_2$.

As an example consider the function `english` which maps natural numbers to their english name as a string, and `roman` which maps them to their roman

numberals as a string then clearly the mediating morphism is just the function which maps a natural number to the pair consisting of the english spelling of the number as the first componant and the roman numerals as the second.

The mediating morphism of the two functions is written $\langle f1, f2 \rangle$. In Felix, like this:

```
\langle f1, f2 \rangle
```

and the projections are actual functions:

```
proj 0 of string * string
proj 1 of string * string
```

so that the following function always returns `true`:

```
fun check[A,B,C] (f1:C->A, f2: C->B, c:C) =>
  let med = ⟨f1, f2⟩ in
    (proj 0 of A * B) (med c) == f1 c and
    (proj 1 of A * B) (med c) == f2 c
;
fun square(x:int) => x * x;
fun cube(x:int) => x * x * x;
println$ check(square, cube, 42);
```

3.8 Products of Categories

Given two categories A and B there is an obvious "set theoretic" product $A \times B$ consisting of all pairs of objects (a, b) and with arrows also being paired up: an arrow between (a, b) and (c, d) is just a pair of arrows $(f : a \to c, g : b \to d)$. This is known as *parallel composition.*

In Felix, you can write this kind of code:

```
fun f(x:int):long =>x.long + 1l;
fun g(x:double):string => x.str;
var h = \prod(f,g);
println$ h (1,42.7);
```

where the `\prod` operator which renders as $\prod$ converts a tuple of functions to a function of tuples.

3.9 Functors

This leads to the following vital idea, that you can have maps between categories. A function from one category to another maps each object of the domain category to an object of the codomain category, and each arrow of the domain to an arrow of the codomain, subject to the requirement that the maps of objects must agree with the maps on identities. This requirement comes from the fact that in a category the identies and objects are really the same thing.

Now a map can satisfy a much stronger requirement: that it be *structure preserving.* This means triangles map to (possibly degenerate) triangles.

Definition 7. Let A and B be categories, and $F : A \to B$ be a map from A to B, then F is called a *functor* if

$$F(a1 \circ_A a2) = F(a1) \circ_B F(a2)$$

In other words, the composite of two arrows in the domain category maps to the composite of the mapped arrows in the codomain category.

It has been said the *whole purpose of category theory is to define functors.*

Definition 8. If $F : A \to B$ and $G : B \to C$ are functors, we can define a map $H : A \to C$ by

$$H(f) = G(F(f))$$

called the composite of the two functors and written

$$H(f) = G \circ F \ F \odot G$$

Lemma 2. *The composite of two functors is a functor.*

The reason functors are so totally important is this:

Lemma 3 (Scalability)**.** *A collection of categories and functors between them closed under functor composition is itself a category.*

This is the most important idea in all of mathematics and especially in programming. The scalability lemma says that categories are their own meta-language; that the relations between categories is modular and these modules are scalable in that they can be combined to form the same kind of module again.

In programming terms this gives rise to the idea that abstractions themselves can be abstracted, and makes it clear, conversely, that instances of an abstraction are still abstract, at a lower level.

Definition 9. If A and B are categories the collection of all functors from A to B is a category called the *functor category* from A to B. It is written, variously,

$$B^A = \mathbf{Func}(A, B)$$

3.10 Polymorphic Data Types

Now I am going to explain what it means for programming.

A polymorphic data type is precisely a functor.

Consider a C++ data structure like say `std::list`. What does it do? If you have it a data type like `int` it will give you a new data type `list<int>` which is a list of ints! This is exactly what a functor does!

Ok, so how about `std::map`? It has two parameters for the key and value. Well, duh, the domain category is just a product, right!

Data functors usually go from products of **MonoType** to **MonoType**.

Now, we have not covered the important property: structure preservation. What this says for `list` is that if you have a function `f` from `int` to `double`, for example, then there is a function in the image of the functor from `list<int>` to `list<double>` where if the i-th element of the first list is a_i then the i-th element of the second list will be $f(a_i)$.

This justifies saying the function f can be applied directly to the list, meaning, it makes a list of the applications of the function to each element. In most languages, we write this as `map f`. In Haskell it is `fmap` which is the sole method of type class `Functor`. In C++, however, there is no mapping function for `list`, instead, you do it using iterators and the `transform` algorithm.

```
// empty list
list<int> int_list = list<int> {1,2,3,4};

// result list
list<double> double_list = list<double>();

// mapping function
double f(int x) { return double (x + 42); }

// do the mapping
transform (int_list.begin(), int_list.end(), f, double_list.begin());
```

3.11 Tuples

Most programmers know what a tuple is even if they just wish their programming language had them. C and C++ do not have tuples!

We know already a tuple is a product. The projections are numbered, in Felix from 0 up. But how do we *make* a tuple?

Of course you know, a pair is a binary tuple with a type like `int * long` and the values are just pairs like `(1, 2L)`, so it appears the type constructor is the

usual product symbol $\times$ written in programs with an infix `*`, and the value constructor is the `,` operator comma. But what is this comma??

You guessed it! It's a functor! In particular, for pairs its a *bifunctor* meaning a functor from a product of two categories. In fact:

$$pair : \mathbf{Monotype} \times \mathbf{MonoType} \to \mathbf{MonoType}$$

What does this do? It takes two types and produces a single type. The value constructor takes two values, and produces a single value.

The use of tuples is absolutely critical. Functors with more than one argument do not exist. In programming languages like C, where it appears they do, every function is ALSO a functor! The separate arguments are combined into a tuple which passed to the underlying actual functions.

This is extremely bad language design! The function and tuple construction should be decoupled. You can see this is so in Felix:

```
fun f(x:int, y: double) => x.double + y;
var arg = 1, 42.7;
println$ f arg;
```

See? The function f only takes one argument. We have decoupled the argument construction from the function calling. The function has type:

```
int * double -> double
```

where the domain type is a single type, that happens to be a tuple type.

The decoupling is mandated by the basic design principles of programming: *separation of responsibility,* one of the SOLID principles.

3.11.1 Unit Tuple

Definition 10. The *unit tuple* is a product of nothing. It has no components, and is usually written `()` and has type `unit`. It has no projections.

Units in a category need not be unique, however they are all isomorphic. With sets, any set with one element is a unit.

3.12 Subcategories

Definition 11. A *subcategory* S of a category A is any collection of objects and arrows which form a category.

Definition 12. There is a natural map called the *subcategory embedding* from S into A written

$$S \hookrightarrow A$$

which takes each object of S to the same object of A, and each arrow of S to the same arrow of A.

Lemma 4. *The subcategory embedding is a functor.*