

# Category Theory for Programmers

Bartosz Milewski

September 2017



This work is licensed under a Creative Commons  
Attribution-ShareAlike 4.0 International License  
(CC BY-SA 4.0). Based on a work at <https://bartoszmilewski.com>.

PDF compiled by Igal Tabachnik,  
based on the work at <https://github.com/sarabander/sicp-pdf>.

# Preface

For some time now I've been floating the idea of writing a book about category theory that would be targeted at programmers. Mind you, not computer scientists but programmers – engineers rather than scientists. I know this sounds crazy and I am properly scared. I can't deny that there is a huge gap between science and engineering because I have worked on both sides of the divide. But I've always felt a very strong compulsion to explain things. I have tremendous admiration for Richard Feynman who was the master of simple explanations. I know I'm no Feynman, but I will try my best. I'm starting by publishing this preface – which is supposed to motivate the reader to learn category theory – in hopes of starting a discussion and soliciting feedback.

**I** WILL ATTEMPT, in the space of a few paragraphs, to convince you that this book is written for you, and whatever objections you might have to learning one of the most abstract branches of mathematics in your “copious spare time” are totally unfounded.

My optimism is based on several observations. First, category theory is a treasure trove of extremely useful programming ideas. Haskell

programmers have been tapping this resource for a long time, and the ideas are slowly percolating into other languages, but this process is too slow. We need to speed it up.

Second, there are many different kinds of math, and they appeal to different audiences. You might be allergic to calculus or algebra, but it doesn't mean you won't enjoy category theory. I would go as far as to argue that category theory is the kind of math that is particularly well suited for the minds of programmers. That's because category theory — rather than dealing with particulars — deals with structure. It deals with the kind of structure that makes programs composable.

Composition is at the very root of category theory — it's part of the definition of the category itself. And I will argue strongly that composition is the essence of programming. We've been composing things forever, long before some great engineer came up with the idea of a subroutine. Some time ago the principles of structural programming revolutionized programming because they made blocks of code composable. Then came object oriented programming, which is all about composing objects. Functional programming is not only about composing functions and algebraic data structures — it makes concurrency composable — something that's virtually impossible with other programming paradigms.

Third, I have a secret weapon, a butcher's knife, with which I will butcher math to make it more palatable to programmers. When you're a professional mathematician, you have to be very careful to get all your assumptions straight, qualify every statement properly, and construct all your proofs rigorously. This makes mathematical papers and books extremely hard to read for an outsider. I'm a physicist by training, and in physics we made amazing advances using informal reasoning. Mathematicians laughed at the Dirac delta function, which was

made up on the spot by the great physicist P. A. M. Dirac to solve some differential equations. They stopped laughing when they discovered a completely new branch of calculus called distribution theory that formalized Dirac's insights.

Of course when using hand-waving arguments you run the risk of saying something blatantly wrong, so I will try to make sure that there is solid mathematical theory behind informal arguments in this book. I do have a worn-out copy of Saunders Mac Lane's *Category Theory for the Working Mathematician* on my nightstand.

Since this is category theory *for programmers*, I will illustrate all major concepts using computer code. You are probably aware that functional languages are closer to math than the more popular imperative languages. They also offer more abstracting power. So a natural temptation would be to say: You must learn Haskell before the bounty of category theory becomes available to you. But that would imply that category theory has no application outside of functional programming and that's simply not true. So I will provide a lot of C++ examples. Granted, you'll have to overcome some ugly syntax, the patterns might not stand out from the background of verbosity, and you might be forced to do some copy and paste in lieu of higher abstraction, but that's just the lot of a C++ programmer.

But you're not off the hook as far as Haskell is concerned. You don't have to become a Haskell programmer, but you need it as a language for sketching and documenting ideas to be implemented in C++. That's exactly how I got started with Haskell. I found its terse syntax and powerful type system a great help in understanding and implementing C++ templates, data structures, and algorithms. But since I can't expect the readers to already know Haskell, I will introduce it slowly and explain everything as I go.



If you're an experienced programmer, you might be asking yourself: I've been coding for so long without worrying about category theory or functional methods, so what's changed? Surely you can't help but notice that there's been a steady stream of new functional features invading imperative languages. Even Java, the bastion of object-oriented programming, let the lambdas in C++ has recently been evolving at a frantic pace — a new standard every few years — trying to catch up with the changing world. All this activity is in preparation for a disruptive change or, as we physicist call it, a phase transition. If you keep heating water, it will eventually start boiling. We are now in the position of a frog that must decide if it should continue swimming in increasingly hot water, or start looking for some alternatives.

One of the forces that are driving the big change is the multicore revolution. The prevailing programming paradigm, object oriented programming, doesn't buy you anything in the realm of concurrency and

parallelism, and instead encourages dangerous and buggy design. Data hiding, the basic premise of object orientation, when combined with sharing and mutation, becomes a recipe for data races. The idea of combining a mutex with the data it protects is nice but, unfortunately, locks don't compose, and lock hiding makes deadlocks more likely and harder to debug.

But even in the absence of concurrency, the growing complexity of software systems is testing the limits of scalability of the imperative paradigm. To put it simply, side effects are getting out of hand. Granted, functions that have side effects are often convenient and easy to write. Their effects can in principle be encoded in their names and in the comments. A function called `SetPassword` or `WriteFile` is obviously mutating some state and generating side effects, and we are used to dealing with that. It's only when we start composing functions that have side effects on top of other functions that have side effects, and so on, that things start getting hairy. It's not that side effects are inherently bad – it's the fact that they are hidden from view that makes them impossible to manage at larger scales. Side effects don't scale, and imperative programming is all about side effects.

Changes in hardware and the growing complexity of software are forcing us to rethink the foundations of programming. Just like the builders of Europe's great gothic cathedrals we've been honing our craft to the limits of material and structure. There is an unfinished gothic **cathedral in Beauvais**, France, that stands witness to this deeply human struggle with limitations. It was intended to beat all previous records of height and lightness, but it suffered a series of collapses. Ad hoc measures like iron rods and wooden supports keep it from disintegrating, but obviously a lot of things went wrong. From a modern perspective, it's a miracle that so many gothic structures had been success-



Ad hoc measures preventing the Beauvais cathedral from collapsing.

fully completed without the help of modern material science, computer modelling, finite element analysis, and general math and physics. I hope future generations will be as admiring of the programming skills we've been displaying in building complex operating systems, web servers, and the internet infrastructure. And, frankly, they should, because we've done all this based on very flimsy theoretical foundations. We have to fix those foundations if we want to move forward.

# 1

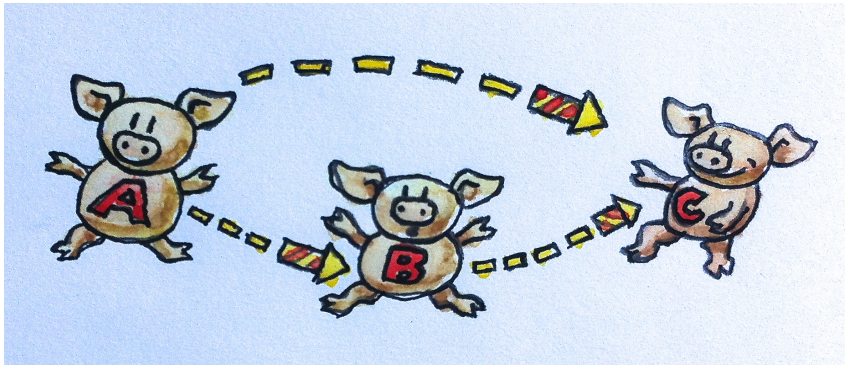
## Category: The Essence of Composition

A CATEGORY is an embarrassingly simple concept. A category consists of *objects* and *arrows* that go between them. That's why categories are so easy to represent pictorially. An object can be drawn as a circle or a point, and an arrow... is an arrow. (Just for variety, I will occasionally draw objects as piggies and arrows as fireworks.) But the essence of a category is *composition*. Or, if you prefer, the essence of composition is a category. Arrows compose, so if you have an arrow from object A to object B, and another arrow from object B to object C, then there must be an arrow - their composition - that goes from A to C.

### Arrows as Functions

Is this already too much abstract nonsense? Do not despair. Let's talk concretes. Think of arrows, which are also called *morphisms*, as functions. You have a function  $f$  that takes an argument of type A and returns a B. You have another function  $g$  that takes a B and returns a





In a category, if there is an arrow going from A to B and an arrow going from B to C then there must also be a direct arrow from A to C that is their composition. This diagram is not a full category because it's missing identity morphisms (see later).

C. You can compose them by passing the result of  $f$  to  $g$ . You have just defined a new function that takes an A and returns a C.

In math, such composition is denoted by a small circle between functions:  $g \circ f$ . Notice the right to left order of composition. For some people this is confusing. You may be familiar with the pipe notation in Unix, as in:

```
ls | grep Chrome
```

Or the chevron  $>>$  in F#, which both go from left to right. But in mathematics and in Haskell functions compose right to left. It helps if you read  $g \circ f$  as “g after f.”

Let's make this even more explicit by writing some C code. We have one function  $f$  that takes an argument of type  $A$  and returns a value of type  $B$ :

```
B f(A a);
```

and another:

```
C g(B b);
```

Their composition is:

```
C g_after_f(A a)
{
    return g(f(a));
}
```

Here, again, you see right-to-left composition:  $g(f(a))$ ; this time in C.

I wish I could tell you that there is a template in the C++ Standard Library that takes two functions and returns their composition, but there isn't one. So let's try some Haskell for a change. Here's the declaration of a function from  $A$  to  $B$ :

```
f :: A -> B
```

Similarly:

```
g :: B -> C
```

Their composition is:

$g \cdot f$

Once you see how simple things are in Haskell, the inability to express straightforward functional concepts in C++ is a little embarrassing. In fact, Haskell will let you use Unicode characters so you can write composition as:

$g \circ f$

You can even use Unicode double colons and arrows:

$f :: A \rightarrow B$

So here's the first Haskell lesson: Double colon means "has the type of..." A function type is created by inserting an arrow between two types. You compose two functions by inserting a period between them (or a Unicode circle).

## Properties of Composition

There are two extremely important properties that the composition in any category must satisfy.

1. Composition is associative. If you have three morphisms,  $f$ ,  $g$ , and  $h$ , that can be composed (that is, their objects match end-to-end), you don't need parentheses to compose them. In math notation this is expressed as:

$$h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$$

In (pseudo) Haskell:

```
f :: A -> B
g :: B -> C
h :: C -> D
h . (g . f) == (h . g) . f == h . g . f
```

(I said "pseudo," because equality is not defined for functions.)

Associativity is pretty obvious when dealing with functions, but it may be not as obvious in other categories.

2. For every object A there is an arrow which is a unit of composition. This arrow loops from the object to itself. Being a unit of composition means that, when composed with any arrow that either starts at A or ends at A, respectively, it gives back the same arrow. The unit arrow for object A is called  $\text{id}_A$  (*identity* on A). In math notation, if f goes from A to B then

$$f \circ \text{id}_A = f$$

and

$$\text{id}_B \circ f = f$$

When dealing with functions, the identity arrow is implemented as the identity function that just returns back its argument. The implementation is the same for every type, which means this function is universally polymorphic. In C++ we could define it as a template:

```
template<class T> T id(T x) { return x; }
```

Of course, in C++ nothing is that simple, because you have to take into account not only what you're passing but also how (that is, by value, by reference, by const reference, by move, and so on).

In Haskell, the identity function is part of the standard library (called Prelude). Here's its declaration and definition:

```
id :: a -> a
id x = x
```

As you can see, polymorphic functions in Haskell are a piece of cake. In the declaration, you just replace the type with a type variable. Here's the trick: names of concrete types always start with a capital letter, names of type variables start with a lowercase letter. So here `a` stands for all types.

Haskell function definitions consist of the name of the function followed by formal parameters — here just one, `x`. The body of the function follows the equal sign. This terseness is often shocking to newcomers but you will quickly see that it makes perfect sense. Function definition and function call are the bread and butter of functional programming so their syntax is reduced to the bare minimum. Not only are there no parentheses around the argument list but there are no commas between arguments (you'll see that later, when we define functions of multiple arguments).

The body of a function is always an expression — there are no statements in functions. The result of a function is this expression — here, just `x`.

This concludes our second Haskell lesson.

The identity conditions can be written (again, in pseudo-Haskell) as:

```
f . id == f
id . f == f
```

You might be asking yourself the question: Why would anyone bother with the identity function – a function that does nothing? Then again, why do we bother with the number zero? Zero is a symbol for nothing. Ancient Romans had a number system without a zero and they were able to build excellent roads and aqueducts, some of which survive to this day.

Neutral values like zero or id are extremely useful when working with symbolic variables. That's why Romans were not very good at algebra, whereas the Arabs and the Persians, who were familiar with the concept of zero, were. So the identity function becomes very handy as an argument to, or a return from, a higher-order function. Higher order functions are what make symbolic manipulation of functions possible. They are the algebra of functions.

To summarize: A category consists of objects and arrows (morphisms). Arrows can be composed, and the composition is associative. Every object has an identity arrow that serves as a unit under composition.

## **Composition is the Essence of Programming**

Functional programmers have a peculiar way of approaching problems. They start by asking very Zen-like questions. For instance, when designing an interactive program, they would ask: What is interaction? When implementing Conway's Game of Life, they would probably ponder about the meaning of life. In this spirit, I'm going to ask: What is

programming? At the most basic level, programming is about telling the computer what to do. “Take the contents of memory address  $x$  and add it to the contents of the register EAX.” But even when we program in assembly, the instructions we give the computer are an expression of something more meaningful. We are solving a non-trivial problem (if it were trivial, we wouldn’t need the help of the computer). And how do we solve problems? We decompose bigger problems into smaller problems. If the smaller problems are still too big, we decompose them further, and so on. Finally, we write code that solves all the small problems. And then comes the essence of programming: we compose those pieces of code to create solutions to larger problems. Decomposition wouldn’t make sense if we weren’t able to put the pieces back together.

This process of hierarchical decomposition and recombination is not imposed on us by computers. It reflects the limitations of the human mind. Our brains can only deal with a small number of concepts at a time. One of the most cited papers in psychology, *The Magical Number Seven, Plus or Minus Two*, postulated that we can only keep  $7 \pm 2$  “chunks” of information in our minds. The details of our understanding of the human short-term memory might be changing, but we know for sure that it’s limited. The bottom line is that we are unable to deal with the soup of objects or the spaghetti of code. We need structure not because well-structured programs are pleasant to look at, but because otherwise our brains can’t process them efficiently. We often describe some piece of code as elegant or beautiful, but what we really mean is that it’s easy to process by our limited human minds. Elegant code creates chunks that are just the right size and come in just the right number for our mental digestive system to assimilate them.

So what are the right chunks for the composition of programs? Their surface area has to increase slower than their volume. (I like this

analogy because of the intuition that the surface area of a geometric object grows with the square of its size – slower than the volume, which grows with the cube of its size.) The surface area is the information we need in order to compose chunks. The volume is the information we need in order to implement them. The idea is that, once a chunk is implemented, we can forget about the details of its implementation and concentrate on how it interacts with other chunks. In object-oriented programming, the surface is the class declaration of the object, or its abstract interface. In functional programming, it's the declaration of a function. (I'm simplifying things a bit, but that's the gist of it.)

Category theory is extreme in the sense that it actively discourages us from looking inside the objects. An object in category theory is an abstract nebulous entity. All you can ever know about it is how it relates to other object – how it connects with them using arrows. This is how internet search engines rank web sites by analyzing incoming and outgoing links (except when they cheat). In object-oriented programming, an idealized object is only visible through its abstract interface (pure surface, no volume), with methods playing the role of arrows. The moment you have to dig into the implementation of the object in order to understand how to compose it with other objects, you've lost the advantages of your programming paradigm.

## Challenges

1. Implement, as best as you can, the identity function in your favorite language (or the second favorite, if your favorite language happens to be Haskell).
2. Implement the composition function in your favorite language. It takes two functions as arguments and returns a function that



is their composition.

3. Write a program that tries to test that your composition function respects identity.
4. Is the world-wide web a category in any sense? Are links morphisms?
5. Is Facebook a category, with people as objects and friendships as morphisms?
6. When is a directed graph a category?

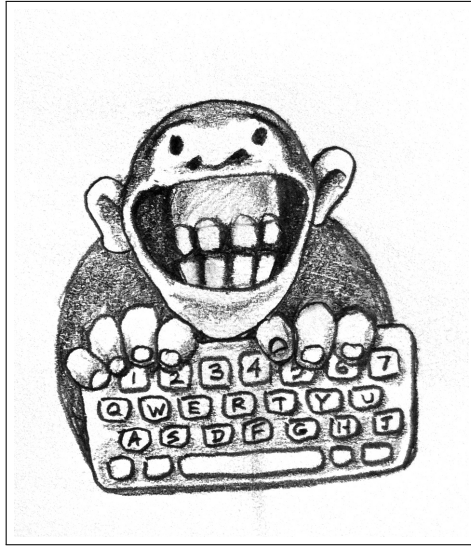
# 2

## Types and Functions

**T**HE CATEGORY OF TYPES AND FUNCTIONS plays an important role in programming, so let's talk about what types are and why we need them.

### Who Needs Types?

There seems to be some controversy about the advantages of static vs. dynamic and strong vs. weak typing. Let me illustrate these choices with a thought experiment. Imagine millions of monkeys at computer keyboards happily hitting random keys, producing programs, compiling, and running them.



With machine language, any combination of bytes produced by monkeys would be accepted and run. But with higher level languages, we do appreciate the fact that a compiler is able to detect lexical and grammatical errors. Lots of monkeys will go without bananas, but the remaining programs will have a better chance of being useful. Type checking provides yet another barrier against nonsensical programs. Moreover, whereas in a dynamically typed language, type mismatches would be discovered at runtime, in strongly typed statically checked languages type mismatches are discovered at compile time, eliminating lots of incorrect programs before they have a chance to run.

So the question is, do we want to make monkeys happy, or do we want to produce correct programs?

The usual goal in the typing monkeys thought experiment is the production of the complete works of Shakespeare. Having a spell checker

and a grammar checker in the loop would drastically increase the odds. The analog of a type checker would go even further by making sure that, once Romeo is declared a human being, he doesn't sprout leaves or trap photons in his powerful gravitational field.

## Types Are About Composability

Category theory is about composing arrows. But not any two arrows can be composed. The target object of one arrow must be the same as the source object of the next arrow. In programming we pass the results on one function to another. The program will not work if the target function is not able to correctly interpret the data produced by the source function. The two ends must fit for the composition to work. The stronger the type system of the language, the better this match can be described and mechanically verified.

The only serious argument I hear against strong static type checking is that it might eliminate some programs that are semantically correct. In practice, this happens extremely rarely and, in any case, every language provides some kind of a backdoor to bypass the type system when that's really necessary. Even Haskell has `unsafeCoerce`. But such devices should be used judiciously. Franz Kafka's character, Gregor Samsa, breaks the type system when he metamorphoses into a giant bug, and we all know how it ends.

Another argument I hear a lot is that dealing with types imposes too much burden on the programmer. I could sympathize with this sentiment after having to write a few declarations of iterators in C++ myself, except that there is a technology called *type inference* that lets the compiler deduce most of the types from the context in which they are used. In C++, you can now declare a variable `auto` and let the compiler figure out its type.

In Haskell, except on rare occasions, type annotations are purely optional. Programmers tend to use them anyway, because they can tell a lot about the semantics of code, and they make compilation errors easier to understand. It's a common practice in Haskell to start a project by designing the types. Later, type annotations drive the implementation and become compiler-enforced comments.

Strong static typing is often used as an excuse for not testing the code. You may sometimes hear Haskell programmers saying, "If it compiles, it must be correct." Of course, there is no guarantee that a type-correct program is correct in the sense of producing the right output. The result of this cavalier attitude is that in several studies Haskell didn't come as strongly ahead of the pack in code quality as one would expect. It seems that, in the commercial setting, the pressure to fix bugs is applied only up to a certain quality level, which has everything to do with the economics of software development and the tolerance of the end user, and very little to do with the programming language or methodology. A better criterion would be to measure how many projects fall behind schedule or are delivered with drastically reduced functionality.

As for the argument that unit testing can replace strong typing, consider the common refactoring practice in strongly typed languages: changing the type of an argument of a particular function. In a strongly typed language, it's enough to modify the declaration of that function and then fix all the build breaks. In a weakly typed language, the fact that a function now expects different data cannot be propagated to call sites. Unit testing may catch some of the mismatches, but testing is almost always a probabilistic rather than a deterministic process. Testing is a poor substitute for proof.

## What Are Types?

The simplest intuition for types is that they are sets of values. The type `Bool` (remember, concrete types start with a capital letter in Haskell) is a two-element set of `True` and `False`. Type `Char` is a set of all Unicode characters like `'a'` or `'ą'`.

Sets can be finite or infinite. The type of `String`, which is a synonym for a list of `Char`, is an example of an infinite set.

When we declare `x` to be an `Integer`:

```
x :: Integer
```

we are saying that it's an element of the set of integers. `Integer` in Haskell is an infinite set, and it can be used to do arbitrary precision arithmetic. There is also a finite-set `Int` that corresponds to machine type, just like the C++ `int`.

There are some subtleties that make this identification of types and sets tricky. There are problems with polymorphic functions that involve circular definitions, and with the fact that you can't have a set of all sets; but as I promised, I won't be a stickler for math. The great thing is that there is a category of sets, which is called `Set`, and we'll just work with it. In `Set`, objects are sets and morphisms (arrows) are functions.

`Set` is a very special category, because we can actually peek inside its objects and get a lot of intuitions from doing that. For instance, we know that an empty set has no elements. We know that there are special one-element sets. We know that functions map elements of one set to elements of another set. They can map two elements to one, but not one element to two. We know that an identity function maps each element of a set to itself, and so on. The plan is to gradually forget all this information and instead express all those notions in purely categorical terms, that is in terms of objects and arrows.

In the ideal world we would just say that Haskell types are sets and Haskell functions are mathematical functions between sets. There is just one little problem: A mathematical function does not execute any code – it just knows the answer. A Haskell function has to calculate the answer. It’s not a problem if the answer can be obtained in a finite number of steps – however big that number might be. But there are some calculations that involve recursion, and those might never terminate. We can’t just ban non-terminating functions from Haskell because distinguishing between terminating and non-terminating functions is undecidable – the famous halting problem. That’s why computer scientists came up with a brilliant idea, or a major hack, depending on your point of view, to extend every type by one more special value called the bottom and denoted by `_|_`, or Unicode  $\perp$ . This “value” corresponds to a non-terminating computation. So a function declared as:

```
f :: Bool -> Bool
```

may return `True`, `False`, or `_|_`; the latter meaning that it would never terminate.

Interestingly, once you accept the bottom as part of the type system, it is convenient to treat every runtime error as a bottom, and even allow functions to return the bottom explicitly. The latter is usually done using the expression `undefined`, as in:

```
f :: Bool -> Bool
f x = undefined
```

This definition type checks because `undefined` evaluates to bottom, which is a member of any type, including `Bool`. You can even write:

```
f :: Bool -> Bool
f = undefined
```

(without the `x`) because the bottom is also a member of the type `Bool->Bool`.

Functions that may return bottom are called partial, as opposed to total functions, which return valid results for every possible argument.

Because of the bottom, you'll see the category of Haskell types and functions referred to as **Hask** rather than **Set**. From the theoretical point of view, this is the source of never-ending complications, so at this point I will use my butcher's knife and terminate this line of reasoning. From the pragmatic point of view, it's okay to ignore non-terminating functions and bottoms, and treat **Hask** as bona fide **Set**<sup>1</sup>.

## Why Do We Need a Mathematical Model?

As a programmer you are intimately familiar with the syntax and grammar of your programming language. These aspects of the language are usually described using formal notation at the very beginning of the language spec. But the meaning, or semantics, of the language is much harder to describe; it takes many more pages, is rarely formal enough, and almost never complete. Hence the never ending discussions among language lawyers, and a whole cottage industry of books dedicated to the exegesis of the finer points of language standards.

There are formal tools for describing the semantics of a language but, because of their complexity, they are mostly used with simplified academic languages, not real-life programming behemoths. One such tool called *operational semantics* describes the mechanics of program execution. It defines a formalized idealized interpreter. The semantics of

---

<sup>1</sup>Nils Anders Danielsson, John Hughes, Patrik Jansson, Jeremy Gibbons, **Fast and Loose Reasoning is Morally Correct**. This paper provides justification for ignoring bottoms in most contexts.



industrial languages, such as C++, is usually described using informal operational reasoning, often in terms of an “abstract machine.”

The problem is that it’s very hard to prove things about programs using operational semantics. To show a property of a program you essentially have to “run it” through the idealized interpreter.

It doesn’t matter that programmers never perform formal proofs of correctness. We always “think” that we write correct programs. Nobody sits at the keyboard saying, “Oh, I’ll just throw a few lines of code and see what happens.” We think that the code we write will perform certain actions that will produce desired results. We are usually quite surprised when it doesn’t. That means we do reason about programs we write, and we usually do it by running an interpreter in our heads. It’s just really hard to keep track of all the variables. Computers are good at running programs – humans are not! If we were, we wouldn’t need computers.

But there is an alternative. It’s called *denotational semantics* and it’s based on math. In denotational semantics every programming construct is given its mathematical interpretation. With that, if you want to prove a property of a program, you just prove a mathematical theorem. You might think that theorem proving is hard, but the fact is that we humans have been building up mathematical methods for thousands of years, so there is a wealth of accumulated knowledge to tap into. Also, as compared to the kind of theorems that professional mathematicians prove, the problems that we encounter in programming are usually quite simple, if not trivial.

Consider the definition of a factorial function in Haskell, which is a language quite amenable to denotational semantics:

```
fact n = product [1..n]
```

The expression `[1..n]` is a list of integers from 1 to `n`. The function `product` multiplies all elements of a list. That's just like a definition of factorial taken from a math text. Compare this with C:

```
int fact(int n) {
    int i;
    int result = 1;
    for (i = 2; i <= n; ++i)
        result *= i;
    return result;
}
```

Need I say more?

Okay, I'll be the first to admit that this was a cheap shot! A factorial function has an obvious mathematical denotation. An astute reader might ask: What's the mathematical model for reading a character from the keyboard or sending a packet across the network? For the longest time that would have been an awkward question leading to a rather convoluted explanation. It seemed like denotational semantics wasn't the best fit for a considerable number of important tasks that were essential for writing useful programs, and which could be easily tackled by operational semantics. The breakthrough came from category theory. Eugenio Moggi discovered that computational effect can be mapped to monads. This turned out to be an important observation that not only gave denotational semantics a new lease on life and made pure functional programs more usable, but also shed new light on traditional programming. I'll talk about monads later, when we develop more categorical tools.

One of the important advantages of having a mathematical model for programming is that it's possible to perform formal proofs of cor-

rectness of software. This might not seem so important when you're writing consumer software, but there are areas of programming where the price of failure may be exorbitant, or where human life is at stake. But even when writing web applications for the health system, you may appreciate the thought that functions and algorithms from the Haskell standard library come with proofs of correctness.

### **Pure and Dirty Functions**

The things we call functions in C++ or any other imperative language, are not the same things mathematicians call functions. A mathematical function is just a mapping of values to values.