In defence of programming

Richard Bornat
inaugural lecture, delivered 22nd January 2004, corrected and revised

April 13, 2005

I have to confess that I'm nervous. Although I've been confidently lecturing for more than 30 years and been on soapboxes for even longer, I've never been allowed before to talk to a captive audience of specialists and non-specialists, to try to explain to them why I love my work and to try to convert them to love it too. After this evening I'll probably never get the chance again. I'm so nervous that I've even written a script, which goes against all my principles. These bat-wings make me even more nervous ...

I've decided to enjoy myself by telling you that programming is truth, truth is beauty, and beauty fun. The video you've just been watching is of me on Aldeburgh beach, dancing the proof of a three-line program which I'll show you later on. We'll come to that proof later: it is beautiful, fun and yet to be invented. The music[1] just seemed right.

1 Does programming need defending?

Programming is a huge industry; Microsoft bestrides the world like a colossus; university computing is booming, and every computing department offers a programming course. I've even been appointed Professor of Computer Programming. Can programming need defence?

Let me begin with an illustration. On 16th May 2003, a vice-chairman of the Conservative Party attacked the standards of university education on Any Questions in these terms:

> ... we now have, to quote a Labour minister in charge of higher education, we now have Mickey Mouse courses such as learning how to groom a horse or golf management whereby you can do a three [year] university course writing essays on these subjects. That, ladies and gentlemen, is education, education, education. [CLAPPING]

This is the kind of knockabout fun which passes for the politics of university funding. Never mind that there are no courses solely on horse grooming; the public, the press and evidently the audience

[1]The Be Good Tanyas, *The littlest birds (sing the prettiest songs).*

that evening all believe that there are. They also believe that Middlesex and places like it offer such courses.

(In case you think I'm making it all up, I'm quoting from the BBC's online transcription of the discussion. I heard Michael Howard talking in the same terms last month. And in case you aren't frightened of Michael Howard ...)

After a few minutes discussion about the world of Mickey Mouse, the presenter tried to bring the discussion back to reality:

> DIMBLEBY: Sorry one more thing - are you aware of hundreds, maybe thousands of people who want to groom horses going to university in order to study how to groom a horse?
>
> PANELLIST: Jonathan, Jonathan, Jonathan, right now in some universities you have a drop out rate of 40 per cent, those people are clearly unsuited to have gone into university in the first place, they are going to leave without a degree and with some sort of debt. What we are trying to do is to make sure that that 40 per cent don't end up there but instead do something that is more of a vocational training - carpenters, joiners, electricians, plumbers, IT consultants, programmers and so on. So that's what we are saying is to weed out those people who aren't going to last the course.

There we are! It's official: programmers are hewers of wood and drawers of water, doomed dropouts, ill-equipped for the intellectual rigours of university life.

My thesis is that programming is not at the bottom of the intellectual pyramid, but at the top. It's creative design of the highest order. It isn't monkey or donkey work; rather, as Edsger Dijkstra famously claimed, it's amongst the hardest intellectual tasks ever attempted. William Morris railed against the separation between design and labour imposed by the factory system in the 19th century. Programming is pure design; it is our first chance to establish an industry in which workers can delight in their labours.

If we can liberate programming from the factory system everybody will benefit. A celebration run by designers gave us the Festival Hall, a beautiful building still in use that will outlive us all, and a festival that lost money. Modern process management techniques, fifty years later, gave us the Millenium Experience, an empty dome and lost far more money. The factory system doesn't suit design: it flattens, crumples, tears and bends it and covers it with slime from the bottom of the dustbin. Modern factory-produced software is a blemish on history, an insult to our intelligence. Last month I sat next to somebody who was reading a three-inch thick manual, part of his training on how to *use* a popular spreadsheet. Programs can be beautiful appliances which elegantly extend the range of activities we can carry out, effortlessly amplify our powers. Factory design drags us down, makes us pay the costs of our own oppression. Colleagues, relations and friends, I call you to rebellion!

Most shocking of all is that many academics agree with the industrialist who told me recently that programming ought to be reduced to mindless work "at the level of MCSE": bricklaying rather than architecture. This despite the experience over decades, across all universities and all countries, that programming is so hard it's almost impossible to teach. Something between 30% and 50% of computing students never really learn to program and never have. Students will always vote with their feet against the hardest courses, but surely we should neither pave their way nor shoe their feet. Perhaps programming ought not to be studied at university, but that would be because it's too hard, not because it's "vocational".

Programming is not boring; it is not mindless; it is hard but it's beautiful and it's fun. I begin my defence.

2 Introduction: history

It's now more than half a century since information joined material and energy as a driver of production and the programming revolution began. The seminal collision between mathematical logic, which became programming, and the electro-mechanical calculator, which became the computer, happened sometime during the horrors of the Second World War, probably in Bletchley. After 1945 demobilised hardware engineers scrambled to exploit their secret knowledge, sometimes using their service passes to raid Army or Post Office depots for electronic parts. The Manchester University Baby won the race by a whisker in 1948, and a commercial version appeared in 1951. The architecture of the electronic digital computer – for various reasons called after the Hungarian-American mathematician John von Neumann – was already settled. It isn't seriously challenged to this day: specialist designs have often overtaken it, but somehow the original has always found a way back.

Demobilised mathematicians weren't idle. Stored programs are part of the von Neumann architecture, a consequence of Turing's 1930s mathematical insights which he took first to Bletchley and then to Manchester. Church, also in the 1930s, had made programs into mathematical formulae. The 1948 Baby was already a universal computing engine, able to imitate any computer that you could describe with a program – which as Babbage glimpsed and Turing proved, was any computer at all provided the computer had enough memory. Every computer since has been the same; universal computing engines gave us word processors, spreadsheets, graphical interfaces and everything else. The first formula translators emerged in the mid 1950s, programs which could translate other programs from one notation to another, spawning the idea of programming languages.

By the end of the 1950s universality was really biting, and the programming tail was wagging the computer dog. Programming became big business. Hardware became less and less specialised, more and more designed to support the 'execution' – following, carrying out – of programs. Although, for legal and technical reasons, the income was still in hardware for a few decades yet, the costs and the benefits were in programming. Certainly, programming wouldn't exist were there not a computer industry, but programming is in the saddle, not under it. Software, as the linguistically inventive USAnians describe it, does things with hardware that the hardware can't do for itself.

Now, over fifty years later and in a world which has ceded power to the software industry – power which it wields irresponsibly, I'm sorry to say – few people not in this room or in rooms like this realise what has happened. Programs are written in mathematical logic: every single one is a sketch of a proof of a mathematical theorem. Every programmer is a mathematical wizard. Not a very powerful wizard, because most of the theorems are extremely boring, and more of an apprentice than a sorcerer because most of the sketches are more than somewhat mistaken – i.e. most programs have bugs. Bugs apart, though, programmers do real magic, making working machinery out of mathematical abstractions. Useful things are made out of material which is less than thin air, less substantial even than a vacuum. It's a turning in of mathematics on itself which mathematicians invented, with Frege and Russell, and then abandoned, when Gödel made it seem like a dead end. Out of the rubble that the mathematicians left behind there grew an infinite tree of knowledge in whose branches programmers may sit and play for ever. Some of the branches are a bit dirty but it's not too hard to get out from under Microsoft if you try.

The triumph of software over hardware is so complete that when you interact with a 'computer', you are always interacting with a program. You wouldn't want to talk to the hardware, believe me! – it talks in binary numerals, for one thing, and it is completely unforgiving of the slightest mistake. Programming has driven a digital revolution which has subverted or taken over television, radio, music telephones, letter-writing, book-writing, technical drawing, clothes washing, motoring, flying, hospital intensive care, university planning, and so on and on. Programs running on general-purpose hardware replace specially-crafted analogue machinery like typewriters and electronic oscillators and vastly expand the range of things that can be done.

As ever, progress is by no means all a good thing. To give an example close to home, the vast managerial planning apparatus which blights the life of everybody in university life from the lowliest lecturer to the highest Vice-Chancellor, so-called Quality so-called Assurance, would be impossible were there not computer programs on every desk driving printers in every corridor. We promised you a paperless office – I'm part of the generation which made that promise – and instead we dropped a paper mountain on your head.

I joined the party in the treetops in 1965, as soon as I realised that the physics that I'd enjoyed so much at school was unlike the science that I barely endured at university but was instead very much like programming. I've been lucky ever since to have been employed to write programs, to argue about programs, to try to help other people to learn programming, to reason about programs, even to talk about programming today. Apart from a disgraceful interval when my ambition sat on my common sense and I tried to be an academic manager, I've been riding bicycles while I think about programs and that's the way I want it to stay.

3 Design and materials

To be a designer, you must know your material. Chairs are made of wood, houses are made of bricks, tall buildings are made of concrete. Concrete chairs are possible but a bit daft; brick chairs break if you lean backwards; wooden tower blocks won't go very high; wooden houses don't look like concrete or brick houses. Program designers have to know what you can and can't do with programs, and that's a lot to know. Once you've made your choices of materials there's nothing left to do, because "what you can do with a program" is so difficult to describe, you have to build the thing.

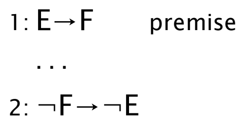
Here's a folding bicycle. Anybody will recognise that it's a miracle of design. It isn't a unique creation: the way it folds goes back at least half a century to the commando bikes of WW2, and there are several different designs on sale today that look rather similar. It's got a huge aluminium tube for lightness and stiffness, so it's a modern technological miracle too. It's beautiful because it's arguably the best that can be done at the moment. When we know how to make one which is smaller and lighter and faster, all at the same time, we'll do it and this one will look quaint. It's a splendid advertisement for Form Follows Function (I'm a modernist at heart).

Here's a personal digital assistant. It too is a miracle of design. The bicycle design is almost all on the outside, all visible except for the hub gear. The PDA design is almost all inside, almost entirely invisible except for its physical size – it's small enough to go in a shirt pocket or the corner of a handbag, and that's an important part of the design. It's not what it seems: it's really a huge program which imitates an address book, a diary, a to-do list and all sorts of other things which you can find out about if you really need to. The physical object is small but the program is millions and millions of lines of text. The design is invisible on purpose; the programmer's version of Form Follows Function is the Invisible Appliance. Our ideal is to make something which you use without knowing you are using it. We can sometimes make things which you can use without reading the manual. The PDA comes close to the ideal of pick-up-and-use.

Although smart graphical designers have been involved, and although graphical design is almost its only visible non-physical component, it's a safe bet that nobody drew a sketch of the display and said "you geeks build me something that looks like that". The design evolved, starting out with the discovery that you could do some of the tasks the PDA does on huge mainframes, then on minicomputers, then on personal computers, then on horrid geeky things that you strapped to your belt if you didn't mind everybody being able to spot you a hundred yards away, finally disappearing into a pocket-sized object and hiding from sight so that its owners can be spotted as trendy, not geeky.

Here's a program that I designed with a colleague – Bernard Sufrin, who is in the audience and with whom I've conspired all my academic life. I decided that I'd like to make a symbolic calculator to teach logical proof, having seen various ones that didn't work very well. Bernard straightened out my ideas and in the first few weeks of our collaboration set out an architecture that has lasted over a decade. Then I fixed his mistakes and he fixed mine and so it went on, until Bernard turned his attention to other things and I had to fix my own mistakes. I think I can claim that all the effort was worthwhile, and that Bernard and I made a large step towards a tool that allows natural logical proof. I've used the latest version to teach a version of logic called Natural Deduction. A year or so ago I modified the program to help teach disproof, the discovery of counter-examples to invalid logical claims.

Here's a proof in Jape of the theorem which says that absence of an effect is enough to prove non-operation of the cause. E stands for the cause and F the effect; $E \rightarrow F$ is as close as we can get to saying that E causes F (please forgive me if you know enough logic to recognise how drastically I'm simplifying) and the hook means "not" or impossibility. E causes F; therefore impossibility/non-occurrence of F means non-operation of E. Students find the proof in stages, beginning with a statement of the problem on a blank canvas:



```
1: E→F      premise
   ...
2: ¬F→¬E
```

This is what $\neg F \rightarrow \neg E$ means – that $\neg E$ is a logical consequence of $\neg F$:

1:	$E \to F$	premise
2:	$\neg F$	assumption
	$\dots$	
3:	$\neg E$	
4:	$\neg F \to \neg E$	$\to$ intro 2–3

This is what $\neg E$ means – that E isn't allowed, that occurrence of E leads to nonsense:

1:	$E \to F$	premise
2:	$\neg F$	assumption
3:	E	assumption
	$\dots$	
4:	$\bot$	
5:	$\neg E$	$\neg$ intro 3–4
6:	$\neg F \to \neg E$	$\to$ intro 2–5

And then, this is what $E \to F$ means – that when you have E you must also have F:

1:	$E \to F$	premise
2:	$\neg F$	assumption
3:	E	assumption
4:	F	$\to$ elim 1,3
	$\dots$	
5:	$\bot$	
6:	$\neg E$	$\neg$ intro 3–5
7:	$\neg F \to \neg E$	$\to$ intro 2–6

And that is what we predicted – F and $\neg F$ at the same time is the logical definition of nonsense:

```
1: E→F          premise
2: | ¬F |       assumption
3: | | E | |    assumption
4: | | F | |    → elim 1,3
5: | | ⊥ | |    ¬ elim 4,2
6: | ¬E |       ¬ intro 3-5
7: ¬F→¬E        → intro 2-6
```

The idea of Jape was to relieve students of the burden of accuracy, allowing them to concentrate on the problems of search – that is, to teach them proof strategy. This wasn't a new educational idea, and nor is the idea of a program which assists proof development. The two ideas together – a symbolic proof *calculator* which gives its users no hints and no "help", beyond refusal to make illegal steps – was novel, though. I hoped that by doing lots of examples and some reflection, novices would begin to see what logic means, and perhaps even see some of its problems. That didn't happen entirely as I would have wished, and James Aczel of the Open University carried out a research project to test what students really did with Jape. As a result of his insights Jape got a whole lot better, and in some cases students really did begin to learn.

Surprisingly, the converse of the claim I proved – that when absence of an "effect" guarantees absence of a "cause" then there really is a causal relationship – isn't so straightforwardly acceptable. This is when things get a bit wierd, and Jape comes into its own as an impartial logical oracle.

Here's a proof, using what are called "classical" ideas of logic, of the reverse claim. It starts rather like the previous one did, by using the meaning of $E \rightarrow F$:

```
1: ¬F→¬E        premise
2: | E |        assumption
   | ... |
3: | F |
4: E→F          → intro 2-3
```

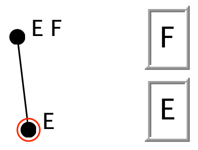
Now the only way we can push the proof forward is to use the "law" of excluded middle, which says that F must be either possible or impossible. If $\neg F$ leads to nonsense, then necessarily F must make sense:

1:	$\neg F \to \neg E$	premise
2:	E	assumption
3:	$\neg F$	assumption
	$\dots$	
4:	$\bot$	
5:	F	contra (classical) 3-4
6:	$E \to F$	→ intro 2-5

and then we can show that the nonsense arrives very much as before:

1:	$\neg F \to \neg E$	premise
2:	E	assumption
3:	$\neg F$	assumption
4:	$\neg E$	→ elim 1,3
5:	$\bot$	¬ elim 2,4
6:	F	contra (classical) 3-5
7:	$E \to F$	→ intro 2-6

That"law" doesn't make sense if you think of F as a proof or a program. It isn't the case, for lots of claims – like "it will rain tomorrow" – that you can always either prove it or disprove it *right now*. If we repeal the "law" we must take an alternative view of logic, in which claims can't be trusted until they are proved and notions of "true" and "false" go out of the window. Then we can give a counter-example. Rather a weasely counter-example, but one none the less. Here it is:



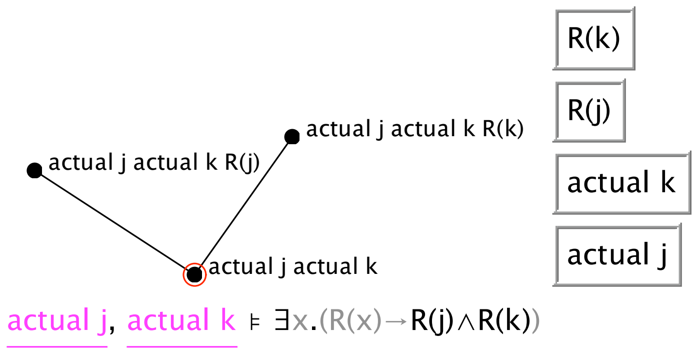
$$\neg F \to \neg E \nvDash E \to F$$

We suppose that we might live in the little red-circled world – that's "here and now" and in that world we have a proof of the formula E (E might be "it's raining", for example). We don't have a proof of F yet, but F isn't impossible (the claim doesn't say that it has to be; F might be "I'm wet") and so we can imagine a future in which we have proofs of both E and F. In this situation, easy to describe and certainly corresponding, for some Es and Fs to the world we live in, we *can't* say that when we have a proof of E we necessarily have a proof of F, because here are we in the red-ringed world with one but not the other. So it's unarguably a counter-example to the claim that E always causes F – unsurprising, because nobody ever thought that such a claim holds no matter what E and F might be, which is what $E \rightarrow F$ says.

The same situation is also an example – this is the weasely bit – for the $\neg F \rightarrow \neg E$ claim. That's because in this little universe you can't claim F is impossible – it does happen, but not just yet. And if you can't claim $\neg F$ then (weasel, weasel) it's reasonable to claim $\neg F \rightarrow G$ for any G at all – on the principle of "if your grandmother had wheels she'd be a bicycle" which is incontrovertible but seems as if it ought to be irrelevant. In particular you *can* claim $\neg F \rightarrow \neg E$, so this situation is a counter-example to the claim written below it – it supports one thing but not the other.

Jape colours and underlines things to help students see what's happening in these controversial disproofs. When they are slightly unconvincing, as in this case, that's part of the educational point: logic isn't a given, it's a construct and it's imperfect.

Here's an even more imperfect example, which I call the universal drunk. The logical claim appears to be that in any universe which has at least two individuals, there must be somebody who, when they are drunk, makes everybody else drunk as well. This is the counter-example:



I'm not going to try to explain what's going on. The educational point is that the claim is quite shocking to 18-year-old students. It seems to imply that you or I could make them drunk if we wanted to. Young people, contrary to tabloid imaginings, are more censorious than libidinous. In fact the claim says one thing to classicists, who accept a proof of their version, and describes the drunk only to constructivists, who are happy to disprove his existence by the diagram I showed. By supporting both kinds of calculation, which it's quite difficult to do accurately, Jape confronts novice computer science students with some important philosophical questions.

Jape is distributed free on the Web. It's about to go open-source, when I get round to publishing the source code. It will be the underpinning of a book that I have to deliver to the publishers in March, and we shall see if the design works without its designer. I haven't tried Jape yet with Middlesex students (hint), and the disproof stuff hasn't been properly evaluated to see what it really does (another hint).

In the end Jape is just an example of a program whose surface is as simple as we can make it and whose complexity is artfully hidden. Its surface polish was buffed up over several years, as Bernard and I found out how to hide more and more of the internal workings and how to make the images look more and more like those you might see in a book or on a blackboard. It can't be a surprise that smart people and evolutionary design will lead to nice results. It might be a surprise that geeks can do it alone, unaided by "design professionals".

It's my contention that programming is the *whole* design process. "I can see how to make a computer do this wonderful thing" is where you start. Design is inspirational, insightful; it uses dreams but it isn't just dreaming. It's absurd to imagine that a technologically-ignorant designer, one who couldn't actually build a program, first has a beautiful idea and then the tunnel-vision geeks are lashed to the oars to row it home. That didn't happen with Jape, it didn't happen with the Palm Pilot, it didn't even happen with the Birdy. Yet it is what many universities teach in so-called 'software engineering' courses – division of labour in the factory system applied to design. You will expect me to say that the result won't often be good design and that it's fairly likely to resemble a pile of poo. (It may comfort you to know that senior computer scientists know that the software you use is causing the difficulties you have when using your computer, and that we don't think it's your fault.)

Beauty in programming isn't independent of utility, it *is* utility. A beautifully designed tool shows you, as soon as you pick it up, how to use it and what it can do. Beautiful programs don't need three-inch thick manuals; they don't need *training* manuals at all.

4 Factory design doesn't work

The oddest thing about the application of factory methods to programming is that everybody knows it doesn't work. You don't have to take my word for it: the factory owners tell you so, every time they sell you something. I can show you part of the licence agreement which I had to agree to when I first began to use my laptop. You will all have clicked "agree" next to something very similar, and probably never read it. Mine's from Apple, but it's industry standard.

First of all they tell you that you didn't really buy what you thought you did:

> 1. **General**. The software, documentation and any fonts accompanying this License whether on disk, in read only memory, on any other media or in any other form (collectively the "Apple Software") are licensed, not sold, to you by Apple Computer, Inc. [...] You own the media on which the Apple Software is recorded but Apple and/or Apple's licensor(s) retain ownership of the Apple Software itself.

Get it? You own a CD, but you don't own even a copy of the software. That's a legal oddity, something to do with the law not accepting that information is stuff, but the distinction between plastic and software is more important than that. The CD at least is guaranteed to work:

> 5. **Limited Warranty on Media**. Apple warrants the media on which the Apple Software is recorded and delivered by Apple to be free from defects in materials and

> workmanship under normal use for a period of ninety (90) days from the date of original retail purchase.

but the software is *not* guaranteed. Quite the contrary – you are warned that it could very well be one of those piles of poo (the underlining is mine, not Apple's):

> 6. **Disclaimer of Warranties**. YOU EXPRESSLY ACKNOWLEDGE AND AGREE THAT USE OF THE APPLE SOFTWARE IS <u>AT YOUR SOLE RISK</u> AND THAT <u>THE ENTIRE RISK</u> AS TO SATISFACTORY QUALITY, PERFORMANCE, ACCURACY AND EFFORT <u>IS WITH YOU</u>. EXCEPT FOR THE LIMITED WARRANTY ON MEDIA SET FORTH ABOVE AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THE APPLE SOFTWARE IS PROVIDED "AS IS", <u>WITH ALL FAULTS AND WITHOUT WARRANTY OF ANY KIND</u> [...]

It goes on and on, in capital letters and with minimal punctuation, for quite some time, expressing the same idea over and over again: this stuff quite possibly doesn't work, and more fool you for trying to buy it.

The maximum extent permitted by applicable law is, well, everything. The law lets them get away with this for the same reason that it lets baked-bean manufacturers underfill some of their cans: nobody really knows how to do any better. Even the open-source movement, quite uncontaminated with the factory system, says something similar in every copyright notice, once again using capital letters so you can't miss the message:

> Jape is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

The best we can say is that the factory system doesn't deliver what its advocates claim it does: quality software at predictable cost. To their credit, the software engineering researchers that I've met at Middlesex – Colin Tully, Darren Dalcher, Manny Lehmann – avoid telling people how to make things and instead, in proper scientific fashion, investigate why 50% of software projects fail across both the public and the private sector. I'm the one who wonders why even the ones that don't fail are so often piles of poo.

5 What I do

I sometimes think that I'm sometimes a good designer, and I've produced three or four reasonable designs over my working life if you count Jape as one. But I've spent most of my time either teaching or down in the engine room with the galley slaves, working on program translators or trying to make tiny bits of program work a bit better. Nowadays my delight is to work on very small programs, wondering what they mean and how we can possibly know.

If we are ever – with or without the factory system – to make programs that are fit for purpose, we will have to understand how they work. That'll be necessary, for example, whenever we have to change our programs because we find out that our initial designs aren't as good as we though. (And that, as Manny Lehmann and E.L. Wisty observed, is always.)

Programs are bits of formal logic, the kind of logic invented by Frege, improved by Russell, and half sunk by Gödel. Every program, if it worked, would be a proof of a theorem. It's an oddity that we normally don't know which theorem. That's the problem of *specification*: how do we say *exactly* what a program does? The problem of *verification* then follows: how do we check that a program corresponds to its specification?

There have been lots of attempts to link programs back to their logical roots. The results can mostly be summed up in the phrase "well yes, but ...". It's darned difficult to specify what programs do and it's harder still to prove that they do it. We're still stuck dealing with smallish programs: the best that's been done is the mathematical development of the safety software for a driverless line in the Paris Metro, a program a few thousand lines long which caused consternation amongst engineers at the train company when it was delivered without any bugs at all. An engineer can't test something that won't break ...

Writing correct software – where we have all three of specification, program and proof – is now possible but expensive. We can only do it for some kinds of designs, and the more precise we try to be the smaller our examples must be. Even if we could do it more easily there would still be a practical and philosophical problem: how can we know that the specification corresponds to the design aims? Correctness in programming won't eliminate the need for evolutionary design; my colleagues have jobs for years to come.

The most practical approach to program-proving so far was developed by Tony Hoare – once Professor at Oxford, now retired to Microsoft Research Cambridge – and that's the tradition I work within. Hoare logic treats programs as logical texts and constructs formal proofs about them using rules in much the same way as the Jape examples that I showed you, though in a special logic designed for the job. Hoare logic and the specification attempts that it inspired was at first so successful that people began to talk, twenty-five years ago, about formal methods of program design and construction. Those methods are unfortunately still being developed, and Hoare is still looking for ways to insert them into the factory design system. But we have made progress, and I've been part of some of the progress that's been made most recently.

One of the problems of Hoare logic has been that it only works for some kinds of program. This is partly on purpose: if you can prove little bits of program separately and put the proofs and programs together without much fuss, you can build up large proofs and large programs without too much work, and only some kinds of programs do that for you. But lots of programs which people like to write, and which we have good reason to believe are perfectly ok, don't fit Hoare's procrustean bed at all and make horribly difficult proofs.

I work with such awkward programs, with a special kind of awkwardness called pointers. Pointers in programming are like addresses in the post ofice. If you know somebody's address you can send them a postcard, ask them for a loan, buy a bicycle from them, all by post. You can give the address to your friends and nothing happens to the addressee, except they might get more postcards. You don't carry around buildings in your filofax, you carry addresses, because addresses weigh less. Pointers are used by almost every program nowadays, for much the same reason of lightness, but they aren't at all easy to deal with in Hoare's original logic.

The problem is that the same pointer can appear in several places, and if you use the pointer to change the contents of the memory location it points to – rather like re-arranging the furniture in a building whose address you know, but ok because it's what pointers are for – you have, in principle, changed the meaning of all the other parts of the program which use the same pointer. Pointers cause linkages between bits of programs which seem to break Hoare's Lego-style rules of proof and program assembly. Until recently, Hoare logic people were sure that pointers were beyond the pale.

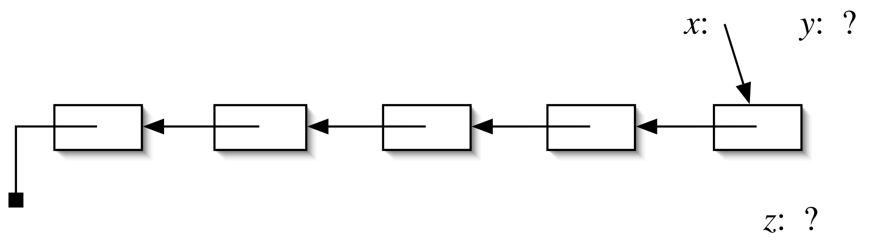
The group I work in – myself, Peter O'Hearn, John Reynolds, Cristiano Calcagno, Hongseok Yang, Ivana Mijajlović, Noah Torp-Smith – is using separation to solve the problems of pointers. Some members, like me, prove programs any way we can; the logicians then think up new logics to do the same thing but more beautifully; then the program provers try to catch up and think of some more proofs in the new logic; and so it goes on. My next target is programs that do packet-switching on the internet, but that's a bit specialised so I'll start back at the beginning.

About thirty years ago Rod Burstall showed how to do a proof of a particular program. This is the program, which is pretty short (I'll explain it later):

```
$y :=$ nil;
while $x \neq$ nil do
   $z := x;\ x := [x];\ [z] := y;\ y := z$
 od
```

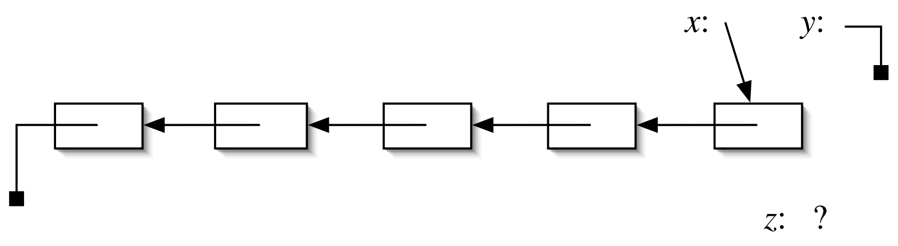
This program can be written on one line but it's perhaps best called a three-liner – initialisation, loop control, loop body. It has a particular property: it reverses a list of cells in a computer's memory without using any extra space (sounds pretty stupid, but that's the sort of thing programs have to do a lot, it's quite hard to do, and it helps to get it right). Programmers have always been confident that they know what the program does: the problem is to find a provable specification and then we'd be *sure* what it does.

It's easy to show what it does in pictures, or with a dance. Some philosophers (and many programmers) might say that we have a proof in the pictures or those movements. Program provers know that pictures often mislead, that pictures of examples are especially misleading, and demand linguistic precision. But the pictures are illuminating, so we'll take the case of a five-element list. To begin we must know that x points to the head of the list, and we have variables y and z whose contents are irrelevant:



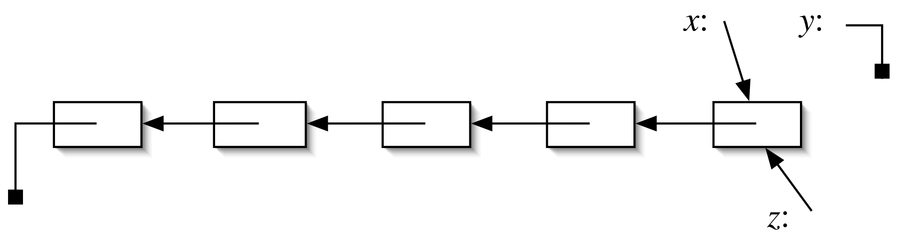
```
y:=nil; while x≠nil do z:=x; x:=[x]; [z]:=y; y:=z od
```

Initialisation overwrites y with the special end-of-list value nil. That makes y point to a list as well – the zero-length "empty" list:



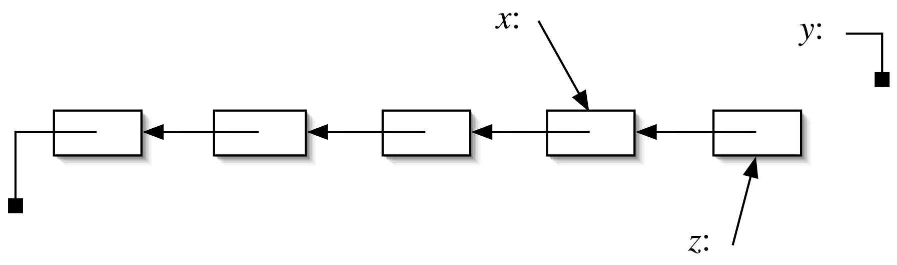
```
y:=nil; while x≠nil do z:=x; x:=[x]; [z]:=y; y:=z od
```

Now because x doesn't contain nil – i.e. because it points to something – we begin the loop. $z := x$ copies the pointer in x and overwrites z:



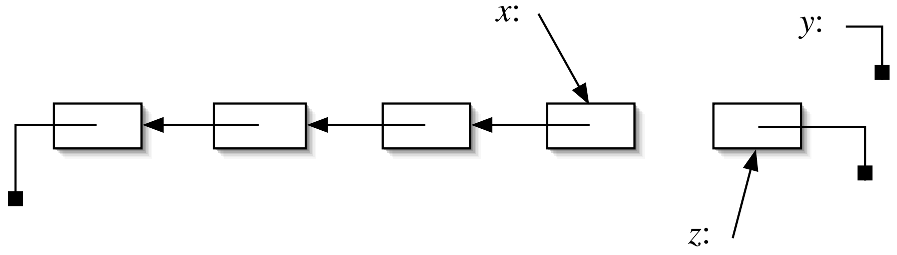
y:=*nil*; *while x*≠*nil do z*:=*x*; *x*:=[*x*]; [*z*]:=*y*; *y*:=*z od*

Next, $x := [x]$ takes what x points to, copies what's there, and overwrites x. The effect is to make x point one place down the list:



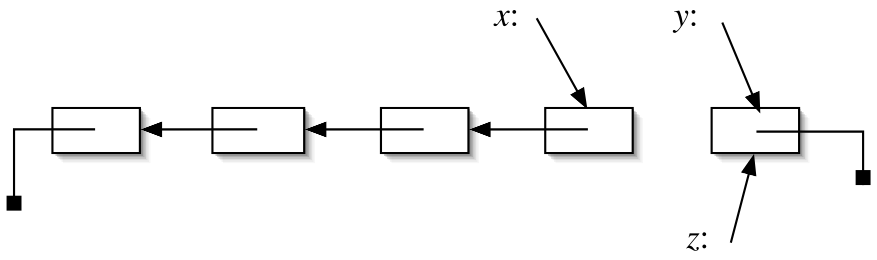
y:=*nil*; *while x*≠*nil do z*:=*x*; *x*:=[*x*]; [*z*]:=*y*; *y*:=*z od*

Then $[z] := y$ copies y and overwrites what z points to. The effect is to put that cell on the front of the y-list:



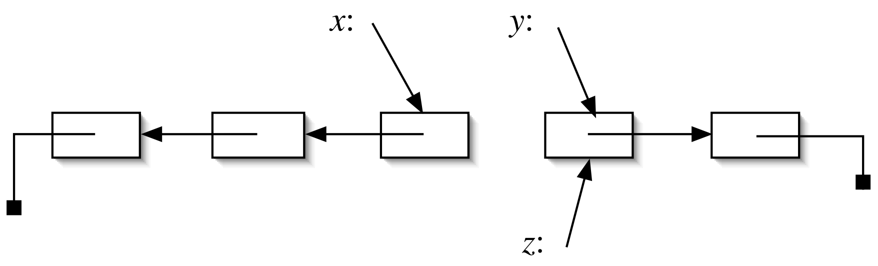
```
y:=nil; while x≠nil do z:=x; x:=[x]; [z]:=y; y:=z od
```

Finally, $y := z$ makes y point where z does:



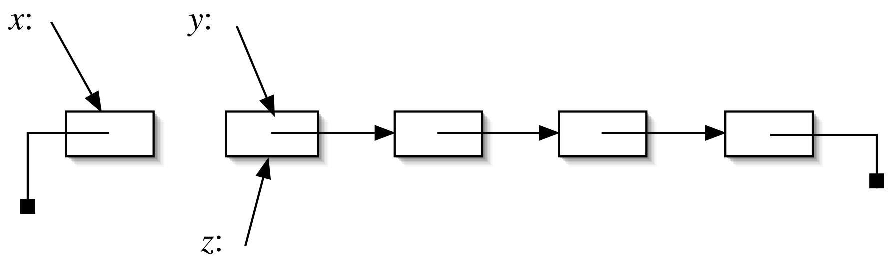
```
y:=nil; while x≠nil do z:=x; x:=[x]; [z]:=y; y:=z od
```

That's one execution of the loop, which moved x down one position and gave the cell it used to point to to y. Because x is not yet nil, we do the whole loop again, and another cell moves from the x-list to the y-list:



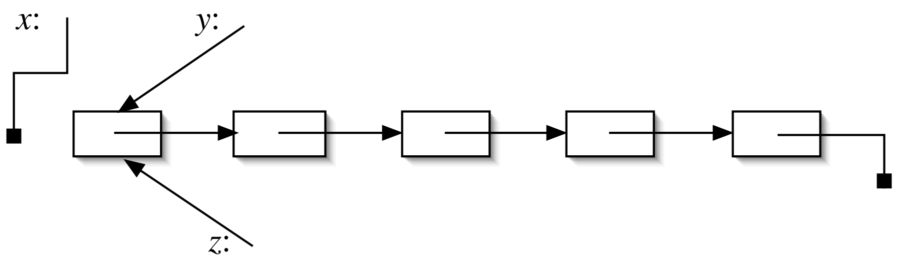
```
y:=nil; while x≠nil do z:=x; x:=[x]; [z]:=y; y:=z od
```

... and so on, till x is pointing at the last thing in the list:



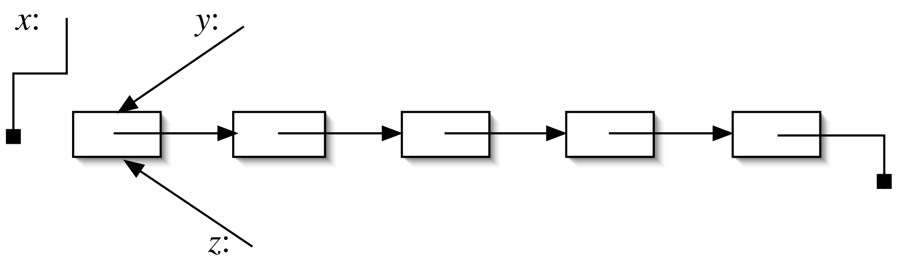
```
y:=nil; while x≠nil do z:=x; x:=[x]; [z]:=y; y:=z od
```

The next execution of the loop gives y what x points to, and gives x what was in $[x]$ which is nil:



```
y:=nil; while x≠nil do z:=x; x:=[x]; [z]:=y; y:=z od
```

And now, because x is at last nil, the loop is finished and we've constructed in y a pointer to a reversed version of the list that x once pointed to:



$$\boxed{y{:=}nil;\ while\ x{\neq}nil\ do\ z{:=}x;\ x{:=}[x];\ [z]{:=}y;\ y{:=}z\ od}$$

It's possible to dance that proof! (Show dance)

What I do is try to construct proofs that are as evocative as the dance, but more general than the pictures. I use separation logic: the contribution that this brings to Hoare logic is to recognise that programs change memory one cell at a time. Every programmer knows, for example, that lists in memory are a sequence of connected but separate cells, ending with (and sometimes consisting only of) the non-pointer nil. (If your mathematics is a little rusty, you may want to look away for a few minutes – I'll tell you when it's over.) We can say what a list is with a simple definition: a zero-length list is represented by nil and takes no space; a list of $n+1$ cells needs a single cell which contains a pointer x' and *separately* ($\star$) a list of n cells pointed to by x':

$$\begin{array}{r}\text{list nil }\langle\,\rangle \mathrel{\hat{=}} \textbf{emp}\\ \text{list } x\ (\langle x\rangle +\!+\ xs) \mathrel{\hat{=}} \exists x'\cdot(x\mapsto x' \star \text{list } x'\ xs)\end{array}$$

With that, I can give you a flavour of the proof of the program. Begin by stating the starting position: x points to a list of indeterminate length:

```
{list $x$ $X$}
$y := \text{nil}$;
while $x \neq \text{nil}$ do
   $z := x;\ x := [x];\ [z] := y;\ y := z$
 od
```

After initialisation y points to a zero-length list:

```
$\{\text{list } x\ X\}$
$y := \text{nil};$
$\{\text{list } x\ X \wedge \text{list } y\ \langle\,\rangle\}$
while $x \neq \text{nil}$ do
   $z := x;\ x := [x];\ [z] := y;\ y := z$
od
```

– and therefore, logically, we can find an α and a β which combine to make a reversed version of X:

```
$\{\text{list } x\ X\}$
$y := \text{nil};$
$\{\text{list } x\ X \wedge \text{list } y\ \langle\,\rangle\}$
$\{\exists \alpha, \beta \cdot (\text{list } x\ \alpha \wedge \text{list } y\ \beta \wedge (\mathit{rev}\ \alpha \mathbin{++} \beta = \mathit{rev}\ X))\}$
while $x \neq \text{nil}$ do
   $z := x;\ x := [x];\ [z] := y;\ y := z$
od
```

That last line is the whole trick of it. The program arranges things so that there is always an α and a β which keep the formula true; the αs keep getting smaller and the βs bigger, and you can't do that for ever (it's like descending a staircase: you are bound to get to the bottom sometime even though you only go one step at a time). Eventually α will be zero-length and we will have found a β which is a reversed version of X.

Every time we go inside the loop we know that x is not nil; therefore α has at least one cell and can be written as $\langle x\rangle \mathbin{++} \alpha'$:

```
$\{\text{list } x\ X\}$
$y := \text{nil};$
$\{\text{list } x\ X \star \text{list } y\ \langle\,\rangle\}$
$\{\exists \alpha, \beta \cdot (\text{list } x\ \alpha \star \text{list } y\ \beta \wedge (\beta \mathbin{++} \mathit{rev}\ \alpha = \mathit{rev}\ X))\}$
while $x \neq \text{nil}$ do
   $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (\mathit{rev}\ (\langle x\rangle \mathbin{++} \alpha') \mathbin{++} \beta = \mathit{rev}\ X)\}$
   $z := x;\ x := [x];\ [z] := y;\ y := z$
od
```

Then we rearrange using a list equivalence: if we reverse a non-empty list the thing at the front goes to the back. In formula terms, $\mathit{rev}\ (\langle x\rangle \mathbin{++} \alpha') = \mathit{rev}\ \alpha' \mathbin{++} \langle x\rangle$:

```
{list $x$ $X$}
$y :=$ nil;
{list $x$ $X \star$ list $y$ $\langle\,\rangle$}
{$\exists \alpha, \beta \cdot$ (list $x$ $\alpha \star$ list $y$ $\beta \wedge (rev\ \alpha +\!\!+\, \beta = rev\ X)$)}
while $x \neq$ nil do
   {$x \mapsto x' \star$ list $x'$ $\alpha' \star$ list $y$ $\beta \wedge (rev\ (\langle x \rangle +\!\!+\, \alpha') +\!\!+\, \beta = rev\ X)$}
   {$x \mapsto x' \star$ list $x'$ $\alpha' \star$ list $y$ $\beta \wedge (rev\ \alpha' +\!\!+\, \langle x \rangle +\!\!+\, \beta = rev\ X)$}
   $z := x;\ x := [x];\ [z] := y;\ y := z$
od
```

The rest of the proof is what mathematicians call tedious – all mechanical calculation. The first instruction does no more than establish $z = x$:

```
{list $x$ $X$}
$y :=$ nil;
{list $x$ $X \star$ list $y$ $\langle\,\rangle$}
{$\exists \alpha, \beta \cdot$ (list $x$ $\alpha \star$ list $y$ $\beta \wedge (rev\ \alpha +\!\!+\, \beta = rev\ X)$)}
while $x \neq$ nil do
   {$x \mapsto x' \star$ list $x'$ $\alpha' \star$ list $y$ $\beta \wedge (rev\ (\langle x \rangle +\!\!+\, \alpha') +\!\!+\, \beta = rev\ X)$}
   {$x \mapsto x' \star$ list $x'$ $\alpha' \star$ list $y$ $\beta \wedge (rev\ \alpha' +\!\!+\, \langle x \rangle +\!\!+\, \beta = rev\ X)$}
   $z := x;$
   {$x \mapsto x' \star$ list $x'$ $\alpha' \star$ list $y$ $\beta \wedge (rev\ \alpha' +\!\!+\, \langle x \rangle +\!\!+\, \beta = rev\ X) \wedge z = x$}
   $x := [x];\ [z] := y;\ y := z$
od
```

– which we rearrange, because we are just about to overwrite x:

```
{list $x$ $X$}
$y :=$ nil;
{list $x$ $X \star$ list $y$ $\langle\,\rangle$}
{$\exists \alpha, \beta \cdot$ (list $x$ $\alpha \star$ list $y$ $\beta \wedge (rev\ \alpha +\!\!+\, \beta = rev\ X)$)}
while $x \neq$ nil do
   {$x \mapsto x' \star$ list $x'$ $\alpha' \star$ list $y$ $\beta \wedge (rev\ (\langle x \rangle +\!\!+\, \alpha') +\!\!+\, \beta = rev\ X)$}
   {$x \mapsto x' \star$ list $x'$ $\alpha' \star$ list $y$ $\beta \wedge (rev\ \alpha' +\!\!+\, \langle x \rangle +\!\!+\, \beta = rev\ X)$}
   $z := x;$
   {$z \mapsto x' \star$ list $x'$ $\alpha' \star$ list $y$ $\beta \wedge (rev\ \alpha' +\!\!+\, \langle z \rangle +\!\!+\, \beta = rev\ X)$}
   $x := [x];\ [z] := y;\ y := z$
od
```

Next we establish $x = x'$ and rearrange:

```
$\{\text{list } x\ X\}$
$y := \text{nil};$
$\{\text{list } x\ X \star \text{list } y\ \langle\,\rangle\}$
$\{\exists \alpha, \beta \cdot (\text{list } x\ \alpha \star \text{list } y\ \beta \wedge (rev\ \alpha +\!\!+\ \beta = rev\ X))\}$
while $x \neq \text{nil}$ do
  $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ (\langle x\rangle +\!\!+\ \alpha') +\!\!+\ \beta = rev\ X)\}$
  $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' +\!\!+\ \langle x\rangle +\!\!+\ \beta = rev\ X)\}$
  $z := x;$
  $\{z \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' +\!\!+\ \langle z\rangle +\!\!+\ \beta = rev\ X)\}$
  $x := [x];$
  $\{z \mapsto x \star \text{list } x\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' +\!\!+\ \langle z\rangle +\!\!+\ \beta = rev\ X)\}$
  $[z] := y;\ y := z$
od
```

We overwrite the cell that z points to:

```
$\{\text{list } x\ X\}$
$y := \text{nil};$
$\{\text{list } x\ X \star \text{list } y\ \langle\,\rangle\}$
$\{\exists \alpha, \beta \cdot (\text{list } x\ \alpha \star \text{list } y\ \beta \wedge (rev\ \alpha +\!\!+\ \beta = rev\ X))\}$
while $x \neq \text{nil}$ do
  $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ (\langle x\rangle +\!\!+\ \alpha') +\!\!+\ \beta = rev\ X)\}$
  $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' +\!\!+\ \langle x\rangle +\!\!+\ \beta = rev\ X)\}$
  $z := x;$
  $\{z \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' +\!\!+\ \langle z\rangle +\!\!+\ \beta = rev\ X)\}$
  $x := [x];$
  $\{z \mapsto x \star \text{list } x\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' +\!\!+\ \langle z\rangle +\!\!+\ \beta = rev\ X)\}$
  $[z] := y;$
  $\{z \mapsto y \star \text{list } x\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' +\!\!+\ \langle z\rangle +\!\!+\ \beta = rev\ X)\}$
  $y := z$
od
```

We notice that z now points to a list $\langle z\rangle +\!\!+\ \beta$, so we rearrange:

```
$\{\text{list } x\ X\}$
$y := \text{nil};$
$\{\text{list } x\ X \star \text{list } y\ \langle\,\rangle\}$
$\{\exists \alpha, \beta \cdot (\text{list } x\ \alpha \star \text{list } y\ \beta \wedge (rev\ \alpha \mathbin{++} \beta = rev\ X))\}$
while $x \neq \text{nil}$ do
   $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ (\langle x\rangle \mathbin{++} \alpha') \mathbin{++} \beta = rev\ X)\}$
   $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' \mathbin{++} \langle x\rangle \mathbin{++} \beta = rev\ X)\}$
   $z := x;$
   $\{z \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' \mathbin{++} \langle z\rangle \mathbin{++} \beta = rev\ X)\}$
   $x := [x];$
   $\{z \mapsto x \star \text{list } x\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' \mathbin{++} \langle z\rangle \mathbin{++} \beta = rev\ X)\}$
   $[z] := y;$
   $\{z \mapsto y \star \text{list } x\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' \mathbin{++} \langle z\rangle \mathbin{++} \beta = rev\ X)\}$
   $\{\text{list } x\ \alpha' \star \text{list } z\ (\langle z\rangle \mathbin{++} \beta) \wedge (rev\ \alpha' \mathbin{++} \langle z\rangle \mathbin{++} \beta = rev\ X)\}$
   $y := z$
od
```

Last we make $y = z$, rearrange slightly, and we've found the new α and β candidates:

```
$\{\text{list } x\ X\}$
$y := \text{nil};$
$\{\text{list } x\ X \star \text{list } y\ \langle\,\rangle\}$
$\{\exists \alpha, \beta \cdot (\text{list } x\ \alpha \star \text{list } y\ \beta \wedge (rev\ \alpha \mathbin{++} \beta = rev\ X))\}$
while $x \neq \text{nil}$ do
   $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ (\langle x\rangle \mathbin{++} \alpha') \mathbin{++} \beta = rev\ X)\}$
   $\{x \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' \mathbin{++} \langle x\rangle \mathbin{++} \beta = rev\ X)\}$
   $z := x;$
   $\{z \mapsto x' \star \text{list } x'\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' \mathbin{++} \langle z\rangle \mathbin{++} \beta = rev\ X)\}$
   $x := [x];$
   $\{z \mapsto x \star \text{list } x\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' \mathbin{++} \langle z\rangle \mathbin{++} \beta = rev\ X)\}$
   $[z] := y;$
   $\{z \mapsto y \star \text{list } x\ \alpha' \star \text{list } y\ \beta \wedge (rev\ \alpha' \mathbin{++} \langle z\rangle \mathbin{++} \beta = rev\ X)\}$
   $\{\text{list } x\ \alpha' \star \text{list } z\ (\langle z\rangle \mathbin{++} \beta) \wedge (rev\ \alpha' \mathbin{++} \langle z\rangle \mathbin{++} \beta = rev\ X)\}$
   $y := z$
   $\{\text{list } x\ \alpha' \star \text{list } y\ (\langle y\rangle \mathbin{++} \beta) \wedge (rev\ \alpha' \mathbin{++} (\langle y\rangle \mathbin{++} \beta) = rev\ X)\}$
od
```

And that's it! We can tie the loop-knot, because we have a replacement for α (it's α') and one for β (it's $\langle y\rangle \mathbin{++} \beta$). We know that the loop won't go on for ever, because α' is exactly one cell smaller than α. And that's a mathematical proof of a program. It's the largest formula I'll show you today – but, believe me, it's pretty small for a program proof. It's much, much smaller than earlier attempts and far, far simpler because I was able to exploit separation: when an instruction affects something separated (by $\star$) from the rest of the formula, you can work on that bit separately and ignore the rest. Because our specifications talk about resources rather than connections, pointers have stopped being dangerous.

The proof is not even as tricky as I made it seem. Using Hoare logic properly, reasoning backwards rather than forwards, makes an easier calculation but the logic is perhaps a little too weird for an inaugural lecture audience.

6 A simpler but more remarkable proof

Now I'm going to take you to the cutting edge of research, and show you a novel solution to a problem that has troubled programmers for over thirty years. The cutting but not the bleeding edge: you won't even feel it.

Computer processors – the chips that do the work inside your laptops, your desktops and your servers – keep getting smaller, cheaper and more powerful. Babbage's analytical engine's "mill" was too expensive to build; commercial versions of the Baby were huge, required vast amounts of power and cost hundreds of thousands of pounds; nowadays they are battery driven, are a few centimetres square and millimetres thick and cost hundreds of pounds; but you can buy two-year old designs, obsolete but perfectly usable, for a few dollars each (in boxes of a thousand!).

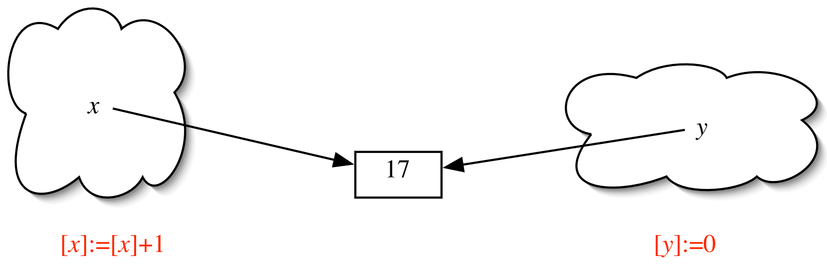
Computers are never fast enough. Programmers see to that: they keep thinking up larger and more intricate problems. The Baby could have driven a modern graphical interface in principle, but it would have been far too slow in practice even if the idea had occurred to anybody. Nowadays we compute planetary orbits and imitate the folding of proteins and simulate nuclear collisions and run program analysers to try to find really well-hidden bugs – all tasks which are just on the edge of feasibility. We need faster computers for today's tasks; we shall need faster ones still for tomorrow's; eventually, perhaps, the hardware engineers will reach the limits of their technology and we'll have to start thinking how we move forward.

You'd think that it would be easy to build a really fast computer by piling lots of cheap processors one on top of the other and getting them to share the calculating task between themselves. Up to a point it is possible, and there are now supercomputers which are assemblies of lots of desktop computers. But they can only run fast on certain rather specialised problems using very complicated programs. The difficulty is that sharing, collaboration and cooperation, which are so easy to arrange and so productive in normal life, are really difficult to establish in a community of mindless slaves – I mean the computers, not the programmers! Everything must be prearranged, and unless you get it perfectly right, the slaves keep colliding and tripping over each others' feet.

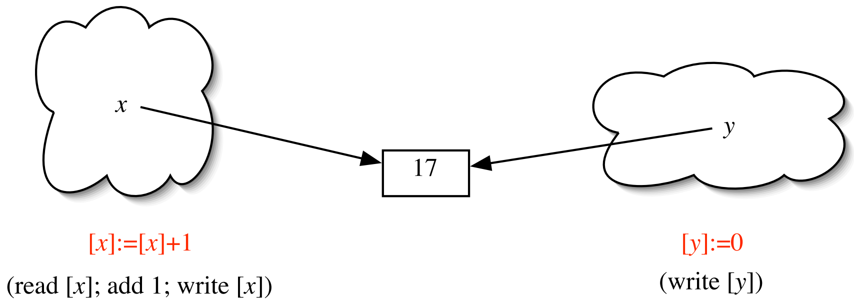
The 20th century computing genius Edsger Dijkstra, who gave us so many insights, recognised, characterised and sketched out a solution to the problem almost 40 years ago. I shan't describe the problem in his terms, and he didn't write the program which I'll show you, but his insight is at the centre of all the solutions we know.

Programs which cooperate have to communicate. The simplest way to communicate in a working community is to write messages on the wall for others to read. But unless they are carefully controlled our thoughtless slaves will overwrite or erase or muddle up each other's messages and read their own messages instead of their partners', and hence cause all sorts of confusion.

Memory cells are the way that computers do message walls. Here's a picture of two program-slaves which each have a pointer to a shared cell. One wants to send a message by adding 1 to the number in the cell; the other wants to send a message by writing 0; by accident this might happen at about the same time.



Now if the actions of the slaves are what Dijkstra called "atomic" – indivisible, effective all at once – then they can't happen at exactly the same time but have to follow each other in one order or the other, and either the cell changes to 18 and then to 0, losing the effect of the +1 message, or it changes first to zero and then to 1, combining both messages. Losing the +1 message is not an error: the 0 message is supposed to overwrite everything so far. The slaves are communicating; the messages are getting through.



But adding 1 to a cell is usually built out of three smaller actions: read to find out what's in the cell; add 1 to what you found; write the result back. Reading and writing are plausibly atomic – it's the way computer memories work for small values – so now there's a new possibility. Read to the left could happen first, then write from the right, then write from the left – we can get the result 18, this time losing the 0 message. This time it is an error: an unlucky slave sent to wipe the blackboard can think he's succeeded but be entirely mistaken.

When our messages are larger than a single number we need more than one cell to hold them, and reads and writes are no longer atomic. Undirected slaves can produce new kinds of confusion: you can read part of one message, part of another, or two slaves writing at about the same time can produce a message which is partly from one and partly from the other. Chaos ensues, and nobody's messages get through.

This kind of collision of idiots – it's called a "race condition" – was the bane of some specialised program designs in Dijkstra's day, and it's the doom of those same designs today and many more besides. When your mouse inexplicably stops moving, when the display freezes, when the modem sticks, when you have to turn off and reboot – in every case the invisible internal slaves were racing and your program's designer bet on the wrong chariot.

If there are no races, there are no collisions. Dijkstra showed us how to eliminate races, by using the technique developed on the railways in the 19th century. The track is divided into blocks about a mile long, with a signal at either end of every block. A train must only go into a block if the signal is green; when it does so the signal changes immediately to red, and it stays red till the train leaves the block by going through the next signal. Block signalling, properly applied, eliminates collisions. Old-fashioned railway signals were wooden arms sticking out from a pole: up to go, across to stop. Dijkstra called his anti-collision devices "semaphores" after those wooden arms.

If we want to communicate via shared memory cells, said Dijkstra, we must set up "critical sections" in our program like the blocks of track, with semaphores at either end. A program must only go into a critical section if the signal is green, and must reset the signal when it leaves; a program which arrives at a signal which is red must wait for it to change. Unlike real railways we can have several critical sections controlled by the same signal, and the effect is that only one program can be in one of those critical sections at the same time. Quite large bits of programs become effectively atomic, in the sense that their effects can't be muddled up with the effects of other critical sections, because there is only ever one program on the track at a time.

Semaphores themselves had to be very special program instructions. Dijkstra explained how to make them, and hardware engineers rapidly obliged. By the mid-1960s every respectable computer had semaphore instructions. But there were problems – organising the waiting round the red signals, for example – and when you wrote your programs safely, programs spent a lot of their time queuing at red signals. People looked for and found ways of speeding things up. It was rapidly recognised that *writing* was the cause of collisions. Any number of slaves could read the current message, provided their reading didn't overlap with the actions of a slave trying to write a new message. "Many readers, only one writer" was a widely-used slogan, and programs got quite a bit faster.

In programming, every solution introduces new problems. Slaves can "starve" if the signalman doesn't treat waiting crowds "fairly". Entire communities can "deadlock", everybody waiting for somebody else to proceed, or "livelock", rushing interminably from red signal to red signal and always just missing their chance. Entire research communities devoted themselves to the problems of cooperation and still do. The problem is that it's very hard to reason about programs which use semaphores (or any of the competing modern mechanisms): you seem to have to take into account the actions of the whole community when reasoning about the actions of a single slave, and that's really difficult.

Because you can't reason about them simply, programmers make mistakes, and mistakes mean bugs. Those bugs are hard to find, because their cause isn't in the action of a single slave, easy to correct, but in the air, in the patterns of cooperation of the whole community. That's why to this day screens freeze, mice die, modems whistle interminably: we don't really know how to do better.

Despite the difficulties, some problems were solved. Thirty years ago Courtois, Heymans, and Parnas published a beautiful solution to the “readers and writers problem”, a program which ensures that readers don’t overlap with writers and writers get exclusive access. It’s a starkly beautiful program because you can’t imagine taking anything away from it; it’s a program which everybody believes works; but it’s an example of a program which nobody knew how to specify so they couldn’t *prove* that it works. P and V are the first letters of Dutch words which mean “lock” and “free”, and are now so famous that everybody forgets that they were once abbreviations.

```
P(read);                        |
count+ := 1;                    |
if count = 1 then P(write);     |
V(read);                        |  P(write);
                                |
... reading happens here ...;   |  ... writing happens here ...
                                |
P(read);                        |  V(write)
count− := 1;                    |
ifcount = 0thenV(write);        |
V(read)                         |
```

The writers have their own block of track, on the right; the readers have three blocks on the left. Actually any code between P(*write*) and V(*write*) is obviously a write track and similarly reading is ok anywhere provided you first register by going through a copy of the start-read track and afterwards hand in your ticket on a copy of the stop-read track. But “obviously” is the source of most of the bugs that have ever been. Luckily, we can now prove this program so there’s no longer any doubt.

(I should be careful at this point. Peter O’Hearn has proposed a novel logical treatment of semaphores. Peter, Josh Berdine, Cristiano Calcagno, Ivana Mijajlović, Uday Reddy, Hongseok Yang and I sat in a trattoria in Venice during a gap in a conference and discussed a logic and a proof in that logic. Cristiano thought of the mathematical model which supports the logic. Hongseok, who thinks up the counter-examples, hasn’t spoken yet. I wrote down the proof. It may sink yet – there are sharp rocks just under the surface of logic – but I’m pretty confident that even if it does we can patch it up and sail on.)

O’Hearn’s treatment of semaphores eliminates the idea of railway tracks and treats them as resource-holders. When you P a semaphore, it opens and releases its resources to you. When you V you have to give back those resources. Since resources are described by formulae, you can give back more or less than you took out. Of course you can’t open the semaphore if it’s already open: just as before, you have to wait. It’s quite unnecessary to change hardware designs: this is just a novel way of looking at what we already have. It means that we don’t need special code-tracks, and we can reason *locally* about what a semaphore does. If a semaphore m holds resource R then in Hoare logic we can always reason $\{\mathbf{emp}\}$ P(m) $\{R\}$ and $\{R\}$ V(m) $\{\mathbf{emp}\}$. That’s an enormous simplification.

The logic we thought up to solve the problem has a special feature. In the proof of the list program

I used $x \mapsto E$ to say that x points to a memory cell which I own and which contains E; if I'd needed to I would have written $x \mapsto \_$ for a cell which I own but whose contents I haven't bothered to read. Since I own the cell I can do whatever I like with it: I can read it, write it, even throw it away. Now we want to distinguish permission to read from permission to write. In the Venice logic $x \mapsto \_$ means that x points to a cell but all I can do is read that cell. I don't own it, and I might be only one of many who have the simultaneous right to read it. I write $x \overset{0}{\longmapsto} \_$ to describe a pointer I do own, which I can read or write, and I can split read permissions from it rather as a flint-knapper splits arrows from a block. $x \overset{n}{\longmapsto} \_$, when $n > 0$, is a block from which I've split exactly n read permissions – I can no longer write, because I would be racing all those readers, but I can still read.

The resource held by the *write* semaphore is a read-write pointer: $x \overset{0}{\longmapsto} \_$. The readers have to steal this away, if they can, when the first reader comes along, and give it back when the last one leaves; in between they have one of those flint blocks, so the *read* semaphore holds (if $count = 0$ then **emp** else $x \xmapsto{count} \_$) – either nothing or a block from which we've split exactly *count* flakes.

(One of the nice things about program-logic research is the freedom it gives you to think. You don't have to use a computer or a laboratory; you don't have to get the cooperation of hardware designers; you don't need anybody's permission to read their programs in quite a new way; you can do it in bath, in the café or on the beach. The only problem is how to stop doing it while you are riding your bicycle, and one day I'll work that out too.)

The proof is now straightforward, only interesting because hardly anybody has seen this logic before. Here is a part of it, the part which explains how the start-read incantation works. It's *much* simpler than the list proof:

```
{$\mathbf{emp}$}
P($read$);
{if $count = 0$ then $\mathbf{emp}$ else $x \xmapsto{count} \_$}
$count+ := 1$;
{if $count = 1$ then $\mathbf{emp}$ else $x \xmapsto{count-1} \_$}
if $count = 1$ then
   {$\mathbf{emp}$}
   P($write$);
   {$x \xmapsto{count-1} \_$}
{$x \xmapsto{count-1} \_$} $\therefore$ {$x \xmapsto{count} \_ \star x \mapsto \_$}
V($read$);
{$x \mapsto \_$}
```

Proofs of this program, of a sort, have existed for a long time. This one is not entirely novel: it uses the insights of others, notably Gries and Owicki's invention of auxiliary variables for use in program logic. But no previous proof is anywhere near as straightforward or as convincing as this one, none can reason locally about the action of a single reader and guarantee just what we want to know with so little effort.

This logical proof is beautiful, because it concisely says so much, but you need to know what you're doing to see that beauty. I know that programs can be beauty all the way down, from the surface all the way down to the tiniest detail. Every tiny detail has to be perfectly judged, or the program won't work.

Those who think this kind of beauty is not part of the "real world" of industry imagine a factory programming system, with obedient code monkeys assembling programs according to blueprints. But if the blueprint is to instruct the monkey perfectly it must be so precise that the assembled parts – semaphores, for example – can't be assembled wrongly, and if that's done, the program is already written and we don't need the monkeys, just the designers. If we merely sketch a design, the code monkeys must know how to select and make correct use of parts, which means they must understand program logic, and then they are designers, not monkeys at all. That's why every attempt to employ programmers as code monkeys has failed and will always fail.

7 What next?

Programming is still mostly mystery; our logic can't yet solve all problems. We can't even say everything I'd like to say about the list-reversal program. Lassoo-shaped lists – ones which start with a straight section but loop back on themselves – are known to be dangerous territory. A program which enters the loop of the lassoo can follow it round for ever and never find its way out. But the list-reversal program reverses the straight section, goes once round the loop of the lassoo reversing that too, then turns back up the straight section, de-reversing and stopping just where it started. I can dance that explanation, as I did on the beach, but I can't find a nice way to prove it. There are probably lots more three, four and five line examples which will intrigue logicians and programmers for years to come.

But we have made great strides and we know enough already to be able to tackle larger problems in the next few years. In particular I hope to make a hole, or at least a serious dent, in proofs of network routing programs, the workhorses which support all e-commerce, e-learning, e-science and e-Uncle-Tom-Cobley-and-all.

8 That's all

That's it. That's what I and those like me have done, do now and will do. We search for beautiful ways of making beautiful things by pure invention. We're supported in what we do by hardware designers; we think we outrank them but they know they could pull the plug on us, so everybody's happy. We're looking for beauty in truth and truth in beauty. We love our work because it's worthy of our love. We're not nerds at all; geeks have wings that perhaps, until today, you couldn't see.

The next time a politician tells you that programmers shouldn't go to university, please tell them where to get off. I'm getting hoarse. Thank you for listening.