Proofs are Programs: 19th Century Logic and 21st Century Computing

Philip Wadler
Avaya Labs

June 2000, updated November 2000

As the 19th century drew to a close, logicians formalized an ideal notion of proof. They were driven by nothing other than an abiding interest in truth, and their proofs were as ethereal as the mind of God. Yet within decades these mathematical abstractions were realized by the hand of man, in the digital stored-program computer. How it came to be recognized that proofs and programs are the same thing is a story that spans a century, a chase with as many twists and turns as a thriller. At the end of the story is a new principle for designing programming languages that will guide computers into the 21st century.

For my money, Gentzen's natural deduction and Church's lambda calculus are on a par with Einstein's relativity and Dirac's quantum physics for elegance and insight. And the maths are a lot simpler. I want to show you the essence of these ideas. I'll need a few symbols, but not too many, and I'll explain as I go along.

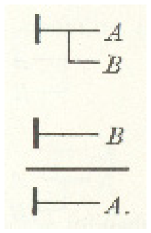
To simplify, I'll present the story as we understand it now, with some asides to fill in the history. First, I'll introduce Gentzen's natural deduction, a formalism for proofs. Next, I'll introduce Church's lambda calculus, a formalism for programs. Then I'll explain why proofs and programs are really the same thing, and how simplifying a proof corresponds to executing a program. Finally, I'll conclude with a look at how these principles are being applied to design a new generation of programming languages, particularly mobile code for the Internet.

1 Gentzen's natural deduction

Aristotle formulated his syllogisms in antiquity, and William of Ockham studied logic in the middle ages. But the discipline of modern logic began with Gottlob Frege's *Begriffschrift*, written in 1879 when Frege was 31.

Ancient logicians attached the name *modus ponens* to a fundamental modes of reasoning: from the premise B *implies* A, and from the premise B, you may draw the conclusion A. For example, let A be the proposition 'we are in Belgium' and B be the proposition 'today is Tuesday'. Given that 'if today is Tuesday then we are in Belgium' and that 'today is Tuesday', you may conclude that 'we are in Belgium'.

Here is how Frege formalized the rule of *modus ponens*.



This figure consists of three *judgements*. Each judgement is indicated by a vertical stroke at the left, indicating that what follows is asserted to be true. There are two judgements above the horizontal

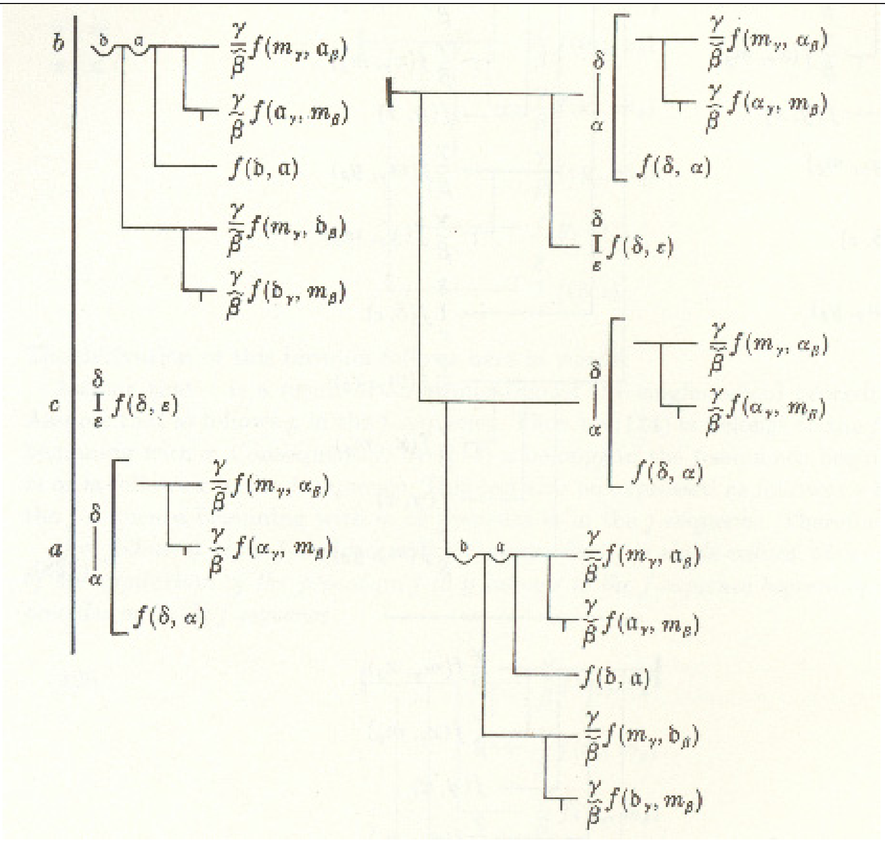


Figure 1: An extract from Frege's *Begriffschrift*, 1879

line, called the *premises*, and one below, called the *conclusion*. Frege used a hook shape to indicate implication. The first premise asserts B *implies* A in Frege's notation, the second premise asserts B, and the conclusion asserts A. Starting from this simple principle, Frege formulated a complex and powerful notation, a sample of which is shown in Figure 1.

Frege's work became the starting point for numerous new formulations of logic. These abandoned his use of pictures, and instead wrote $B \to A$ for B *implies* A and wrote $\vdash A$ for the judgement that asserts A (the $\vdash$ symbol evolved from the stroke Frege wrote to the left of his judgements). In the new notation, *modus ponens* was written like this.

$$\frac{\vdash B \to A \qquad \vdash B}{\vdash A}$$

In addition to *modus ponens*, these systems required some *axioms* like the following.

$$\vdash A \to A \tag{1}$$

$$\vdash A \to (B \to A) \tag{2}$$

$$\vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C)) \tag{3}$$

As with *modus ponens*, you can substitute any proposition you want in place of A, B, and C. The first axiom asserts A *implies* A, which means if we know A then we can conclude A, which seems clear enough. The second asserts A *implies* (B *implies* A), which means if you know A and you know B then you can still conclude A, which lets you ignore irrelevant facts. The third is more complex, and I'll leave it for

$$\frac{}{A \vdash A}\;\text{Id}$$

$$\frac{\Gamma,\, B \vdash A}{\Gamma \vdash B \to A}\;{\to}\text{-I} \qquad \frac{\Gamma \vdash B \to A \qquad \Delta \vdash B}{\Gamma,\, \Delta \vdash A}\;{\to}\text{-E}$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma,\, \Delta \vdash A \wedge B}\;{\wedge}\text{-I} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\;{\wedge}\text{-E}_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\;{\wedge}\text{-E}_2$$

Figure 2: Gentzen's natural deduction

the more motivated reader to puzzle out. But, remarkably, from *modus ponens* and the second and third axioms one can derive every truth about implication, including the first axiom.

In addition to A *implies* B, written $A \to B$, logicians considered other logical connectives, such as A *and* B written $A \wedge B$, and A *or* B written $A \vee B$, and *not* A written $\neg A$. For example, $(A \wedge B) \to A$ is read A *and* B *implies* A, which is another way to say that if you know A and you know B, then you can still conclude A. There are also quantifiers, such as *for all* written $\forall$. For example,

$$\forall x.\, \forall y.\, \forall z.\, (x < y \wedge y < z) \to x < z$$

is read '*for all x, y, and z, if x is less than y and y is less than z then x is less than z*'. In what follows, I'll just consider $\to$ and $\wedge$ and ignore the other logical connectives and the quantifiers.

One might hope for a logic of implication based on something more natural than the three axioms above, especially the daunting axiom (3). This was the goal of Gerhard Gentzen's *natural deduction*, introduced in 1934 when Gentzen was 25.

Gentzen generalized Frege's notion of judgement to include *assumptions*. Whereas Frege's judgements (after the notation evolved) look like $\vdash A$, meaning *assert that A is true*, Gentzen's judgements look look like this

$$B_1,\, \ldots,\, B_n \vdash A$$

meaning *assert that A is true on the assumption that $B_1$and$\ldots$andB_n are all true.* The list of assumptions $B_1,\, \ldots,\, B_n$ may be empty, or contain just one proposition, or contain many. The case where it is empty corresponds to Frege's notion of judgement.

Following tradition, we use the greek letters Γ and Δ to stand for lists of propositions, just as we use A, B, C to stand for single propositions. (They are pronounced *gamma* and *delta*.) Order in the lists is immaterial. We form the union of two lists by writing Γ, Δ, and this also removes any duplicates from the list.

Here is *modus ponens* in Gentzen's system.

$$\frac{\Gamma \vdash B \to A \qquad \Delta \vdash B}{\Gamma,\, \Delta \vdash A}$$

The rule again has two premises. The first premise asserts that B *implies* A holds under assumptions Γ, the second premise asserts that B holds under assumptions Δ, and the conclusion asserts that A holds under the combined assumptions Γ and Δ. For example, let Γ be the assumption 'we are on the whirlwind tour' and Δ be the assumption 'yesterday was Monday', and take A and B as before. Given that 'assuming we are on the whirlwind tour, if today is Tuesday then we are in Belgium' and that 'assuming yesterday was Monday, today is Tuesday', you may conclude 'assuming we are on the whirlwind tour and yesterday was Monday, we are in Belgium'.

Some rules of Gentzen's natural deduction are shown in Figure 2. Gentzen's system includes two sorts of rules, *structural* rules and *logical* rules. The figure has one structural rule, Id, and logical rules

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{B \wedge A\} \vdash B \wedge A}\,\text{Id}}{\{B \wedge A\} \vdash A}\,\wedge\text{-E}_2 \qquad \dfrac{\dfrac{}{\{B \wedge A\} \vdash B \wedge A}\,\text{Id}}{\{B \wedge A\} \vdash B}\,\wedge\text{-E}_1}{\{B \wedge A\} \vdash A \wedge B}\,\wedge\text{-I}}{\{\} \vdash (B \wedge A) \to (A \wedge B)}\,\to\text{-I} \qquad \dfrac{\dfrac{}{\{B\} \vdash B}\,\text{Id} \qquad \dfrac{}{\{A\} \vdash A}\,\text{Id}}{\{A,\, B\} \vdash B \wedge A}\,\wedge\text{-I}}{\{A,\, B\} \vdash A \wedge B}\,\to\text{-E}$$

Figure 3: A roundabout proof

for the connectives $\to$ (*implies*) and $\wedge$ (*and*). Each logical rule is classified as an *introduction* rule if the connective appears in the conclusion but not the premises, and as an *elimination* rule if the connective appears in a premise but not in the conclusion. Thus, in the $\to$-I rule, $\to$ appears in the conclusion, and in the $\to$-E rule, $\to$ appears in the first premise.

Rule Id states that from assumption A one can deduce A, a tautology if ever there was one. Rule $\to$-I states that if from assumptions Γ and B one can deduce A, then from assumptions Γ alone one can deduce that B implies A. Rule $\to$-E is our friend *modus ponens*. Rule $\wedge$-I states that if from assumptions Γ one can deduce A and from assumptions Δ one can deduce B, then from combined assumptions Γ and Δ one can deduce both A and B. Rule $\wedge$-E$_1$ states that if from assumptions Γ one can deduce A and B, then from the same assumptions one can deduce just A, and similarly for $\wedge$-E$_2$ and B.

At first glance, these rules may seem confusing, since 'if-then' appears in three different forms ($\to$ for propositions, $\vdash$ for judgements, and the premises and conclusion of inference rules). The genius of Gentzen was to see that adding such bookkeeping simplified the final system — the rules exhibit a pretty symmetry, and no strange beasts like axiom (3) are required. Further, as we shall see, Gentzen's method of bookkeeping meshes perfectly with viewing proofs as programs, a view not discovered until three decades later.

A small proof built from these rules appears in Figure 3. The proof demonstrates a trivial fact, that from assumptions A and B one can deduce $A \wedge B$. But it does so in a roundabout way: it first demonstrates that from no assumptions one can deduce $(B \wedge A) \to (A \wedge B)$, and that from the assumptions A and B one can deduce $B \wedge A$, and then applies modus ponens. One might reasonably expect that such a proof could be simplified.

A major contribution of Gentzen's 1934 paper was the *subformula property*: he showed that any proof of a statement $\Gamma \vdash A$ can be simplified so that the only propositions it mentions are those in Γ, and A, and any parts (subformulas) of these. Any diversions (like using $B \wedge A$ to prove $A \wedge B$) can be simplified away. Simplified proofs were much easier to reason about, and Gentzen used the subformula property to demonstrate a number of key properties, including the consistency of his logic.

Gentzen's 1934 paper actually introduced *two* ways of formulating logic, natural deduction and sequent calculus. What I've shown here is essentially natural deduction, though it borrows some of its form (such as the use of $\vdash$) from sequent calculus. Gentzen preferred natural deduction, but could only prove the subformula property for the sequent calculus. A way to simplify natural deduction proofs directly was eventually discovered, but as we shall see, this took some time.

Ironically, the key to simplifying natural deduction proofs had already appeared in print. Church's first paper on lambda calculus was published in 1932, two years before Gentzen's paper on natural deduction. As we will see, the two works are in remarkably close correspondence, though they were developed for different reasons entirely.

2 Church's lambda calculus

Alonzo Church introduced lambda calculus in 1932 as part of a new formulation of logic. That formulation turned out to be flawed, but Church suspected that lambda calculus might have independent interest. In that first paper he wrote 'There may, indeed, be other applications of the system than its use as a logic.' Prophetic words!

By 1936, Church had realized that lambda terms could be used to express *every* function that could ever be computed by a machine. Independently, at about the same time, Turing wrote the famous paper on the machine that bears his name. It was quickly recognized that the two formulations were equivalent, and Turing came to Princeton to study with Church between 1936 and 1938.

Shortly thereafter Turing returned to Britain. During the war, At Bletchley Park he worked on machines — early proto-computers — designed to break enemy codes. (Had he not succeeded, you and I might be speaking German.) And within a decade, John von Neumann, who was at Princeton and familiar with the work of Church and Turing, wrote his famous note on the architecture of the stored program computer.

Church reduced all calculation to the notion of substitution. Typically, a mathematician might define a function by an equation. If a function f is defined by the equation $f(x) = t$, where t is some term involving x, then the application $f(u)$ yields the value $t[u/x]$, where $t[u/x]$ is the term that results by substituting u for each occurrence of x in t. For example, if $f(x) = x \times x$ then $f(3) = 3 \times 3 = 9$.

Church provided an especially compact way of writing such functions. Instead of saying 'the function f where $f(x) = t$', he simply wrote $\lambda x.\, t$. In his notation, our example function is written $\lambda x.\, x \times x$. The essence of calculation is described by the following *reduction rule*:

$$(\lambda x.\, t)(u) \quad \Rightarrow \quad t[u/x]$$

For example, $(\lambda x.\, x \times x)(3) \Rightarrow 3 \times 3 \Rightarrow 9$. We use $\Rightarrow$ (pronounced *reduces to*) rather than $=$ to indicate the direction in which terms become simpler. A term of the form $\lambda x.\, t$ is called a *lambda abstraction*, and a term of the form $t(u)$ is called an *application*.

When there is more than one reduction to be performed, it does not matter which you do first. Consider the following, where the incrementing function $\lambda y.\, y + 1$ is applied to 2, and then the squaring function $\lambda x.\, x \times x$ is applied to that result.

$$\begin{tikzcd}
& (\lambda x.\, x \times x)((\lambda y.\, y + 1)(2)) \arrow[dl] \arrow[dr] & \\
(\lambda x.\, x \times x)(2 + 1) \arrow[dr] & & ((\lambda y.\, y + 1)(2)) \times ((\lambda y.\, y + 1)(2)) \arrow[dl] \\
& (2 + 1) \times (2 + 1) &
\end{tikzcd}$$

Here the path on the left first reduces the increment (substituting for y), while the path on the right first reduces the square (substituting for x). The path on the right duplicates some work (it must reduce $(\lambda y.\, y+1)(2)$ twice), but it gets to the same answer in the end, namely 9. That the order of reductions is immaterial makes it much easier to manipulate lambda terms. This key result is called the Church-Rosser theorem.

What about functions of more than one argument? Church used an ingenious idea due to Frege: a function of two arguments can be represented by a function of the first argument that returns a function of the second argument. For instance, the function $g(x, y) = x \times x + y \times y$ is represented in lambda calculus as $\lambda x.\, \lambda y.\, x \times x + y \times y$. Corresponding to the computation $g(3, 4) = 3 \times 3 + 4 \times 4 = 25$ one has the following reduction sequence, which first substitutes for x and then substitutes for y.

$$\begin{array}{ll}
& ((\lambda x.\, \lambda y.\, x \times x + y \times y)(3))(4) \\
\Rightarrow & (\lambda y.\, 3 \times 3 + y \times y)(4) \\
\Rightarrow & 3 \times 3 + 4 \times 4 \\
\Rightarrow & 25
\end{array}$$

This trick was eventually christened *currying* after another logician, Haskell Curry.

The clever thing in Church's formulation is that a function can take a function as its argument, or return a function as its result. For example, the composition of two functions is denoted by the lambda term

$$\lambda f.\, \lambda g.\, \lambda z.\, g(f(z))$$

This takes two functions (f and g) as arguments, then returns a function that takes an argument (z), applies the first function to its argument, and then applies the second function to that result. Here is an example where composition is applied to the increment and squaring functions, and the resulting function is applied to the number 2.

$$\begin{array}{ll} & (\lambda f.\, \lambda g.\, \lambda z.\, g(f(z)))(\lambda y.\, y+1)(\lambda x.\, x \times x)(2) \\ \Rightarrow & (\lambda g.\, \lambda z.\, g((\lambda y.\, y+1)(z)))(\lambda x.\, x \times x)(2) \\ \Rightarrow & (\lambda z.\, (\lambda x.\, x \times x)((\lambda y.\, y+1)(z)))(2) \\ \Rightarrow & (\lambda x.\, x \times x)((\lambda y.\, y+1)(2)) \\ \Rightarrow & (2+1) \times (2+1) \\ \Rightarrow & 9 \end{array}$$

After two steps, substituting for f and g, this reduces to the term we saw earlier, which increments 2 and then squares the result.

The above examples made use of numbers and arithmetic, assuming extra reduction rules like $3 \times 3 \Rightarrow 9$. One might also extend lambda calculus with data structures. For instance, we might add the term $\langle t,\, u\rangle$ to build a pair, and the terms $p.\mathit{fst}$ and $p.\mathit{snd}$ to select the first and second fields from a pair, together with the reduction rules

$$\begin{array}{lcl} \langle t,\, u\rangle.\mathit{fst} & \Rightarrow & t \\ \langle t,\, u\rangle.\mathit{snd} & \Rightarrow & u \end{array}$$

For example, here's a lambda term that swaps the elements of a pair.

$$\lambda z.\, \langle z.\mathit{snd},\, z.\mathit{fst}\rangle$$

We can apply this to the pair $\langle y,\, x\rangle$.

$$\begin{array}{ll} & (\lambda z.\, \langle z.\mathit{snd},\, z.\mathit{fst}\rangle)(\langle y,\, x\rangle) \\ \Rightarrow & \langle \langle y,\, x\rangle.\mathit{snd},\, \langle y,\, x\rangle.\mathit{fst}\rangle \\ \Rightarrow & \langle x,\, y\rangle \end{array}$$

First the lambda reduction rule substitutes for the argument, then the pair reduction rules select out the fields. We'll make use of pairs in what follows.

The remarkable thing about the lambda calculus is that these extensions are unnecessary. Church hit on the clever idea of representing a number by a function that takes a function as argument and applies that function the given number of times. Thus, n is represented by $\lambda f.\, \lambda x.\, f(\cdots f(x))$, where f is repeated n times. In effect, the number n is represented by a 'for' loop that applies any given function n times. Church then showed how to define arithmetic operations as yet other lambda terms. Addition and multiplication are pretty easy — they look a lot like function composition — but subtraction turned out to be quite devious. Booleans, pairs, and other structures can be represented by tricks similar to those used to represent numbers. Even recursive functions can be represented, by means of a cunning device.

It was in this way that Church showed that any function on numbers computable by a machine could be represented by a lambda term, built using only lambda abstraction ($\lambda x.\, t$), function application ($t(u)$), and variables (x), plus the notion of reduction ($(\lambda x.\, t)(u) \Rightarrow t[u/x]$). Just as the physical universe appears to be built from a small number of fundamental particles and forces, the computing universe can be built from just three term forms and one reduction rule.

$$\frac{}{x : A \vdash x : A}\,\text{Id}$$

$$\frac{\Gamma,\ x : B \vdash t : A}{\Gamma \vdash \lambda x.\, t : B \to A}\,{\to}\text{-I} \qquad \frac{\Gamma \vdash t : B \to A \qquad \Delta \vdash u : B}{\Gamma,\ \Delta \vdash t(u) : A}\,{\to}\text{-E}$$

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma,\ \Delta \vdash \langle t,\ u\rangle : A \wedge B}\,\wedge\text{-I} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash t.\mathit{fst} : A}\,\wedge\text{-E}_1 \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash t.\mathit{snd} : B}\,\wedge\text{-E}_2$$

Figure 4: Church's typed lambda calculus

3 Typed lambda calculus

Church introduced a typed version of lambda calculus in 1940. His goal was to avoid paradoxes that beset other logics (including Frege's original logic, which fell prey to Russel's paradox). Just as lambda calculus had applications beyond what Church first intended, so too did types. Type systems in languages like Algol and Pascal can catch many programming errors, and the typed lambda calculus plays the same role for functional programming languages.

We write $A \to B$ for the type of a function from an argument of type A to a result of type B, and $A \wedge B$ for the type of a pair where the first component has type A and the second component has type B.

A typing judgement has the form

$$x_1 : B_1,\ \ldots x_n : B_n \vdash t : A$$

meaning *term t has type A on the assumption that variable x_1 has type B_1 and ... variable x_n has type B_n, where x_1 and ... and x_n are the free variables of term t.*

We now use the greek letter Γ and Δ to stand for lists of variable-type pairs, $x : B$. As before, order in the lists is immaterial. We form the union of two lists by writing $\Gamma,\ \Delta$; if a variable x appears in both Γ and Δ then it must be given the same type in both, and the union removes the duplication.

Here is the rule of function application in Church's system.

$$\frac{\Gamma \vdash t : B \to A \qquad \Delta \vdash u : B}{\Gamma,\ \Delta \vdash t(u) : A}$$

The rule again two premises. The first premise asserts that t is a function from type B to type A, under type assumptions Γ, the second premise asserts that u is a term of type B under type assumptions Δ, and the conclusion asserts that $t(u)$ is a term of type A under the combined typing assumptions Γ and Δ. Note that the free variables of $t(u)$ are those of t unioned with those of u.

The rules of Church's typed lambda calculus are shown in Figure 4. There is one rule for terms that are variables. The other rules are grouped by whether they act on function or pair types ($\to$ or $\wedge$), and by whether the rule *introduces* or *eliminates* a value of that type (I or E). Lambda abstraction introduces a function, while application eliminates it; and a pair constructor introduces a pair, while a field selector eliminates it.

Rule Id states that from the assumption that x is of type A, one can deduce that x is of type A, an obvious tautology. Rule $\to$-I states that if under the assumption that variable x has type B one can deduce that term t has type A, then one can deduce that the lambda abstraction $\lambda x.\, t$ is a function from type A to type B. The free variables of t are those in Γ together with x; while the free variables of $\lambda x.\, t$ are those in Γ alone, since the variable x is considered *bound* by the lambda abstraction. Rule $\to$-E is our old friend, function application. Rule $\wedge$-I states that if term t has type A and term u has type B, then the constructor $\langle t,\ u\rangle$ yields a pair of type $A \wedge B$. Again, the free variables of $\langle t,\ u\rangle$ are those of t

$$\dfrac{\dfrac{\Gamma \cup \{x : B\} \vdash t : A}{\Gamma \vdash \lambda x.\, t : B \to A}\,{\to}\text{-I} \qquad \Delta \vdash u : B}{\Gamma \cup \Delta \vdash (\lambda x.\, t)(u) : A}\,{\to}\text{-E} \quad \Rightarrow \quad \Gamma \cup \Delta \vdash t[u/x] : A$$

$$\dfrac{\dfrac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma \cup \Delta \vdash \langle t,\, u\rangle : A \wedge B}\,\wedge\text{-I}}{\Gamma \cup \Delta \vdash \langle t,\, u\rangle.\mathit{fst} : A}\,\wedge\text{-E}_1 \quad \Rightarrow \quad \Gamma \vdash t : A$$

$$\dfrac{\dfrac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma \cup \Delta \vdash \langle t,\, u\rangle : A \wedge B}\,\wedge\text{-I}}{\Gamma \cup \Delta \vdash \langle t,\, u\rangle.\mathit{snd} : B}\,\wedge\text{-E}_2 \quad \Rightarrow \quad \Delta \vdash u : B$$

Figure 5: Reductions preserve type derivations

combined with those of u. Rule $\wedge$-E$_1$ states that if t is a pair of type $A \wedge B$, then the selector $t.\mathit{fst}$ yields the first field of the pair which has type A. The free variables of $t.\mathit{fst}$ are the same as those of t. Similarly for $\wedge$-E$_2$.

There is one rule for each term form, so given a term and the types of its variables (free and bound) the corresponding type derivation is uniquely determined, and vice versa.

How do reductions affect the type of a term? It is easy to see that they leave the type unchanged, as illustrated in Figure 5. Consider the reduction

$$(\lambda x.\, t)(u) \quad \Rightarrow \quad t[u/x]$$

Given a type derivation for the left side it is easy to construct a derivation for the right side, as follows. The type derivation ending in $\Gamma \cup \{x : B\} \vdash t : A$ will have one leaf of the form $\{x : B\} \vdash x : B$ for each occurrence of x in t. If each x in t is replaced by u, then each such leaf is replaced by the derivation $\Delta \vdash u : B$, giving a derivation for $\Gamma \cup \Delta \vdash t[u/x] : A$, as required. Now consider the reductions for pairs.

$$\begin{array}{rcl} \langle t,\, u\rangle.\mathit{fst} & \Rightarrow & t \\ \langle t,\, u\rangle.\mathit{snd} & \Rightarrow & u \end{array}$$

Given type derivations for the left sides it is trivial to construct derivations for the right sides, since a derivation for $\langle t,\, u\rangle$ directly includes derivations for t and u.

In each of the three reduction rules, the derivation on the left consists of an introduction for a type followed by an elimination for the same type. The reduction rule for functions applies when a lambda term introduces a function, which is immediately eliminated by an application; and the reduction rules for pairs apply when a constructor introduces a pair, which is immediately eliminated by a selector. Whenever an introduction rule meets the corresponding elimination rule the two cancel out, like matter meeting anti-matter.

For example, recall the following sequence of reductions.

$$\begin{array}{ll} & (\lambda z.\, \langle z.\mathit{snd},\, z.\mathit{fst}\rangle)(\langle y,\, x\rangle) \\ \Rightarrow & \langle \langle y,\, x\rangle.\mathit{snd},\, \langle y,\, x\rangle.\mathit{fst}\rangle \\ \Rightarrow & \langle x,\, y\rangle \end{array}$$

Figure 6 shows the corresponding type derivations for this sequence. First the adjacent function introduction and elimination cancel out, and each leaf $\{z : B \wedge A\} \vdash z : B \times A$ is replaced by the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{z : B \wedge A\} \vdash z : B \wedge A}\,\text{Id}}{\{z : B \wedge A\} \vdash z.\mathit{snd} : A}\,\wedge\text{-E}_2 \qquad \dfrac{\dfrac{}{\{z : B \wedge A\} \vdash z : B \wedge A}\,\text{Id}}{\{z : B \wedge A\} \vdash z.\mathit{fst} : B}\,\wedge\text{-E}_1}{\{z : B \wedge A\} \vdash \langle z.\mathit{snd},\, z.\mathit{fst}\rangle : A \wedge B}\,\wedge\text{-I}}{\{\} \vdash \lambda z.\, \langle z.\mathit{snd},\, z.\mathit{fst}\rangle : (B \wedge A) \to (A \wedge B)}\,\to\text{-I} \qquad \dfrac{\dfrac{}{\{y : B\} \vdash y : B}\,\text{Id} \qquad \dfrac{}{\{x : A\} \vdash x : A}\,\text{Id}}{\{x : A,\, y : B\} \vdash \langle y,\, x\rangle : B \wedge A}\,\wedge\text{-I}}{\{x : A,\, y : B\} \vdash (\lambda z.\, \langle z.\mathit{snd},\, z.\mathit{fst}\rangle)(\langle y,\, x\rangle) : A \wedge B}\,\to\text{-E}$$

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{y : B\} \vdash y : B}\,\text{Id} \qquad \dfrac{}{\{x : A\} \vdash x : A}\,\text{Id}}{\{x : A,\, y : B\} \vdash \langle y,\, x\rangle : B \wedge A}\,\wedge\text{-I}}{\{x : A,\, y : B\} \vdash \langle y,\, x\rangle.\mathit{snd} : A}\,\wedge\text{-E}_2 \qquad \dfrac{\dfrac{\dfrac{}{\{y : B\} \vdash y : B}\,\text{Id} \qquad \dfrac{}{\{x : A\} \vdash x : A}\,\text{Id}}{\{x : A,\, y : B\} \vdash \langle y,\, x\rangle : B \wedge A}\,\wedge\text{-I}}{\{x : A,\, y : B\} \vdash \langle y,\, x\rangle.\mathit{fst} : B}\,\wedge\text{-E}_1}{\{x : A,\, y : B\} \vdash \langle\langle y,\, x\rangle.\mathit{snd},\, \langle y,\, x\rangle.\mathit{fst}\rangle : A \wedge B}\,\wedge\text{-I}$$

$$\Downarrow$$

$$\dfrac{\dfrac{}{\{x : A\} \vdash x : A}\,\text{Id} \qquad \dfrac{}{\{y : B\} \vdash y : B}\,\text{Id}}{\{x : A,\, y : B\} \vdash \langle x,\, y\rangle : A \wedge B}\,\wedge\text{-I}$$

Figure 6: A reduction sequence with type derivations

for $\{x : A, y : B\} \vdash \langle y,\, x\rangle : B \times A$. This in turn brings the two pair introductions next to the two pair eliminations, and these cancel out leaving the final simplified term.

There is a subtle and surprising consequence of this argument. The reduction for pairs always makes the derivation tree smaller. The reduction for functions may make the derivation larger, since it removes the subtree $\lambda x.\, t$, but may copy the subtree u many times. However, the subtree $\lambda x.\, t$ has type $B \to A$, while the subtree u has type B, so although there are more subtrees their types are simpler. A moment's thought will show that the types cannot keep getting simpler forever, so reduction of typed lambda terms must always terminate. Computer scientists are used to the idea that the halting problem is unsolvable, but typed lambda calculus provides a rich language for which the halting problem is trivially solved.

The flip side of this is that typed lambda calculus must be limited in its expressiveness. In particular, it cannot express recursion. One can add a new term and typing rule for recursion, restoring expressiveness at the cost of losing termination. You pay your money, and you take your choice.

Many type systems have been based on the principles sketched here, including sum types, record types, abstract types, polymorphic types, recursive types, subtype inclusion, object-oriented types, module types, type classes, kinded types, monad types, and linear types. In all these systems, types provide a guide to the design: start from the types, and the terms and reductions follow. The terms follow as introducers and eliminators for the type, and the reductions follow when an introducer meets an eliminator, and type safety follows by showing that each of the reduction rules preserves type derivations. Types have proven to be one of the most powerful organizing principles for programming languages.

4 The Curry-Howard correspondence

By now, you may have noticed that if you ignore the red part of Figure 4, showing Church's typed lambda calculus, what remains is identical to Figure 2, showing Gentzen's natural deduction. Given a type derivation one can easily construct the corresponding proof, and vice versa. And we have already seen that given a term one can construct the corresponding type derivation, and vice versa. Hence terms and proofs are in one-to-one correspondence.

Further, term reduction corresponds to proof simplification. We noted earlier that the proof in Figure 3 was needlessly roundabout. Rather than proving $A \wedge B$ directly from A and B, it first proves $(B \wedge A) \rightarrow (A \wedge B)$ and $B \wedge A$, then uses modus ponens to get the desired result. As we have seen, this proof corresponds to the term $(\lambda z.\, \langle z.snd,\, z.fst\rangle)(\langle y,\, x\rangle)$, and the simplification of the proof corresponds to the reduction sequence shown in Figure 6.

The terms corresponding to proofs have a natural interpretation. The proof of an implication $B \rightarrow A$ is a function, which takes a proof of the anteceedent B into a proof of the consequent A. In the $\rightarrow$-E rule, the function that proves $B \rightarrow A$ is applied to the proof of B to yield a proof of A. Similarly, the proof of a conjunction $A \wedge B$ is a pair of proofs, one for the component A and one for the component B. In the $\wedge$-E$_1$ rule, selection is applied to the pair of proofs $A \wedge B$ to yield a proof of A, and similarly for the $\wedge$-E$_2$ rule and B.

For simplicity, I've only presented a small fragment of Gentzen's logic and Church's type system, but the correspondence extends further. For instance, the logical connective $A \vee B$ (A *or* B) corresponds to a disjoint union type (or variant record) that stores either a value of type A or a value of type B, and an indication of which. However, one important restriction is that the correspondence works best for *intuitionistic* logic, a logic that does not contain the law of the excluded middle, which asserts that $A \vee \neg A$ is true for any proposition A.

Though Gentzen's natural deduction was published in 1934 and Church's typed lambda calculus was published in 1940, the path to spotting this correspondence was a long one. Gentzen could only make proof simplification work for sequent calculus, not natural deduction. In 1956, Dag Prawitz showed how to simplify natural deduction proofs directly. In that same year, Curry and Feys published the definitive work on combinators, a sort of simplified variant of lambda expressions. Curry (the man for whom currying was named) had noted a correspondence between the types of the combinators and the laws of logic as formulated by Hilbert. Finally, in 1969, W. A. Howard put together the results of Curry and Prawitz, and wrote down the correspondence between natural deduction and lambda calculus. Thus, three decades later, it was finally possible to see the work of Gentzen and Church as two sides of the same coin. By the way, Howard's work, though widely influential, was circulated only on mimeographed sheets; it was not published until 1980, in a book honoring Curry on his 80th birthday.

The living proof of the Curry-Howard correspondence is the appearance of other double-barrelled names in the literature on types. Time and again, a logician motivated by logical concerns and a computer scientist motivated by practical concerns have discovered exactly the same type system, usually with the logician getting there first. Most modern functional languages use the Hindley-Milner type system, discovered by the logician Hindley in 1969 and re-discovered by the computer scientist Milner in 1978. And much recent work on type-directed compilers is based on the Girard-Reynolds system, discovered by the logician Girard in 1972 and re-discovered by the computer scientist Reynolds in 1974.

The Curry-Howard correspondence led logicians and computer scientists to develop a cornucopia of new logics based on the correspondence between proofs and programs. In these systems, rather than write a program to compute, say, the integer square root function, one would instead prove a corresponding theorem, such as that for all integers x such that $x \geq 0$, there exists a largest integer y such that $y^2 \leq x$. One could then extract the integer square root function, which given x returns the corresponding y, directly from the proof. Early systems of this sort include Automath, developed by the logician de Bruijn in 1970, and Type Theory, developed by the philosopher Martin Löf in 1982. Later, computer scientists got into the game, with Nuprl, developed by Constable in 1986, the Calculus of Constructions, developed by Coquand and Huet in 1988, and the Lambda Cube, developed by Barendregt in 1991.

The story continues to develop until today. For instance, one might wonder whether the Curry-Howard correspondence could extend to classical logic (adding back in the law of the excluded middle). Indeed it can, as shown by Timothy Griffin in 1990. Or one might wonder whether Gentzen's simplification result for sequent calculus can be applied to give insight into lambda calculus. Indeed it can, as shown by Barendregt and Ghilezan last year.

The close fit between Gentzen's work and Church's, uncovered three decades after the original papers, is quite astonishing. Recall that a confusing aspect of Gentzen's system was that it involved something like 'if-then' in three different forms: $\rightarrow$ for propositions, $\vdash$ for statements, and the premises and conclusion of inference rules. In retrospect, it is easy to see why there are three different levels: $\rightarrow$ corresponds to types, $\vdash$ corresponds to free variables and terms, and inference rules correspond to the way terms are built. Gentzen, by seeking to understand the deep nature of proof, and Church, by seeking to understand the deep nature of computation, created two systems that were in perfect correspondence.

Church and Curry lived to see their abstract theories have an impact on practical programming languages. They were guests of honor at the 1982 conference on Lisp and Functional Programming, both highly bemused by what had been done with their work. (A young graduate student in the throes of finishing my thesis, I was fortunate enough to attend and meet them both.)

Gentzen was not so fortunate. He did not even survive to see Prawitz's method of directly simplifying natural deduction proofs. During the war he took a post at the University of Prague. On 5th May 1945, the citizen's of Prague revolted against the German forces, and held the city until the arrival of the Russian Army. Gentzen was sent to prison by the new local authorities. One of his friends wrote of this time 'He once confided in me that he was really quite contented since now he had at last time to think about a consistency proof for analysis.' But confusion was rife, and the prison conditions were poor. He died in his cell of malnutrition on 4th August 1945.

5 Conclusions

Church's lambda calculus, both typed and untyped, had a profound impact on the development of programming languages. The notion that functions may take functions as arguments and may return functions as results was particularly important. In the 1960's, Christopher Strachey proposed this as a guiding principle for the design of programming languages, under the motto 'functions as first-class citizens'.

Lisp used the keyword 'lambda' to define functions, but its definition of function differs subtly from that of lambda calculus. Languages that took more direct inspiration from lambda calculus include Iswim (Peter Landin, 1966); Scheme, a dialect of Lisp which got 'lambda' right (Guy Steele and Gerald Sussman, 1975); ML, short for 'metalanguage' (Milner, Gordon, and Wadsworth, 1979), later succeed by Standard ML; Miranda (David Turner, 1985); Haskell, named for Haskell Curry (Hudak, Peyton Jones, Wadler and others, 1987); and O'Caml, a french spin-off of ML (Xavier Leroy and others, 1996).

Iswim and Scheme are untyped, but the other languages have type systems based on the Hindley-Milner and Girard-Reynolds systems. Standard ML is noted for its exploration of module types, Haskell for its type classes, and O'Caml for its object-oriented types. Standard ML, Haskell, and O'Caml are all continuing development, and innovation in their type systems is one of the principle directions of research.

Applications built on top of functional languages, and which themselves use type systems in innovative ways, include: Kleisli, a database language for biomedical data (implemented in Standard ML); Ensemble, a library for building distributed applications and protocol stacks (implemented in O'Caml); Lolita, a natural language understanding system (implemented in Haskell); and Xduce, a language for processing XML (implement in O'Caml).

Functional programming and types have also had a significant impact on Java. Guy Steele, one of the three principle designers of Java, got his start with the functional language Scheme. John Rose, another Scheme programmer, introduced lambda terms into Java in the form of 'inner classes', where they are used to implement callbacks in the graphic user interface and the event framework. The security model of Java depends heavily on type safety, spurring much new work in this area, including type systems to enhance security. And the next generation of Java may include generic types, based partly on the Girard-Reynolds type system (see GJ: A Generic Java, Dr Dobbs Journal, February 2000).

Java and Jini provide some support for distributed computing, but it is clear that they are not the last word. A whole new generation of distributed languages is currently under development, including Milner's pi calculus, Cardelli and Gordon's Ambit, and Fournier and Gonthier and other's Join calculus, and Lee and Necula's proof-carrying code. All of these languages draw heavily on the traditions of typed lambda calculus.

Theorem provers Hewlett-Packard's Runway multiprocessor bus underlies the architecture of the HP 9000 line of servers and multiprocessors. Hewlett-Packard applied the HOL (Higher-Order Logic) theorem prover as part of an effort to verify that caching protocol's in Runway did not deadlock. This approach uncovered errors that had not been revealed by several months of simulation.

The Defence Science and Technology Organisation, a branch of the Department of Defence in Australia, is applying the Isabelle theorem prover to verify arming conditions for missile decoys. The system was used to prove, for example, that the missile cannot be armed when the launch safety switch is not enabled. As a side effect of constructing the proof, some potential errors in the code were spotted.

Both HOL and Isabelle are implemented in Standard ML. Standard ML is a descendant of ML, the metalanguage of the groundbreaking LCF theorem prover, which is in turn an ancestor of both HOL and Isabelle. This circle reflects the intertwined history of theorem provers and functional languages. HOL was developed by Gordon, Melham, and others, with versions released in 1988, 1990, and 1998. Isabelle was developed by Paulson, Nipkow, and others, with version released in 1993, 1994, 1998, and 1999.

ML/LCF exploited two central features of functional languages, higher-order functions and types. A proof tactic is a function taking a goal formula to be proved and returning a list of subgoals paired with a justification. A justification, in turn, is a function from proofs of the subgoals to a proof of the goal. A tactical is a function that combined small tactics into larger tactics. The type system was a great boon in managing the resulting nesting of functions that return functions that accept functions. Further, the type discipline ensured soundness, since the only way to create a value of type *Theorem* was by applying a given set of functions, each corresponding to an inference rule. As noted above, the type system Robin Milner devised for ML remains a cornerstone of work in functional languages. ML/LCF was developed by Milner, Gordon, and Wadsworth, with the complete description published in 1979.

HOL and Isabelle are just two of the many theorem provers that draw on the ideas developed in LCF, just as Standard ML is only one of the many languages that draw on the ideas developed in ML. Among others, Coq is implemented in Caml, Veritas in Miranda, Yarrow in Haskell, and Alf, Elf, and Lego in Standard ML again. A recent issue of the *Journal of Functional Programming* was devoted to the interplay between functional languages and theorem provers.

Proof-carrying code Trust is essential in computing. Almost all of the code you run is not code that you yourself have written. How can you know it will do what you intend to do, rather than, say, wipe your hard disk, or e-mail your credit card numbers to a third party? The Internet has brought this problem into sharper focus, as it increases opportunities for downloading programs while it decreases opportunities to build trust via social interaction. Computer viruses in programs hidden in e-mail documents have brought this problem to public attention, but the problem exists for any program you might run on your machine. Further, not only do you have to trust the programmer not to be malicious, you also have to trust him or her not to violate the security policy by mistake. Such mistakes are in fact quite common, as many who have suffered a machine crash can attest.

There are technical means are available to complement the social means of building trust. For instance, an operating system may allocate permission to read or write files to various accounts. (Unix and NT support this, Windows 98 does not). One can then arrange for an untrusted program to be given permission to read or write only a small selection of programs. (Alas, even on systems that can easily support this, it is not always standard practice to do so.) The policy describing which resources may be accessed is called a security policy.

The bedrock of any security policy is limiting the areas of computer memory that a program can read or write. Once a program overwrites the kernel of the operating system, all bets are off. Even more simply, many viruses or worms operate by overwriting a return location on the stack, which in turn is achieved by writing into an array with an index outside the array boundary.

Thus, types can be intimately tied to security. For instance, if a program is passed the address of an array, the type system can ensure that the program always writes to addresses within the array boundary. Strongly typed languages like Java or Modula 3 guarantee such properties, weakly typed languages like C or C++ do not. Java's popularity as an Internet programming language is based in large part on the fact that its design addresses these security issues. The foundation of this is that Java byte codes are verified before they are run — that is, the byte codes form a typed programming language, and the verifier checks the types.

The simplest way to achieve security is to check each operation each time it is performed, but this can be expensive. To reduce the cost, you may wish to check once and for all before running the program that certain operations are guaranteed to be safe every time they are performed — that is, you want to prove that certain operations always conform to your security policy, and that is exactly what a type system does. Often, one uses a combination of both approaches.

For example, when Java indexes an array, the top two slots on the stack contain an index and a pointer to the array. The verifier guarantees that the slot which is supposed to point to an array really does point to an array (a block of storage containing a length followed by the array contents), so there is no need to check this at run-time. However, the code does need to check at run time that the array index is bounded by the array length. (A clever just-in-time compiler may be able to eliminate this check for some loops, such as when the array index is the same as the loop index and the loop index is bounded by the array length.)

Of course, Java is not the only game in town. For some applications one may want the added flexibility of supplying machine code rather than Java byte codes. One method is to simply trust that the code provided is safe, perhaps using cryptographic techniques to verify that the code is supplied by a known party. When executing machine code, is there any way to supplement trust with tests? There is, and the solutions span an alphabet of acronyms.

One is SFI, or Software Fault Isolation. This inserts extra instructions into the machine code, to check that accesses will not exceed set bounds on the memory. SFI introduces extra run-time overhead, and the security policies it implements are cruder than those imposed by a type system. Typically, one can restrict a whole program to access data in a particular region of memory, but not restrict accesses to a particular array to be within bounds. SFI was developed by Wahbe, Lucco, Anderson, and Graham at Berkeley in 1993.

Another is TAL, or Typed Assembly Language. The type system developed for lambda calculus is flexible enough to be applied to a range of other styles of programming languages — and, remarkably, it can even be extended to assembly language. In Java, the byte codes are annotated with types that are checked by the verifier before the code is run. Similarly, in TAL the machine code is annotated with types that are checked by the type checker before the code is run. The difference is that while Java byte codes were especially designed for checking, in TAL the type system is overlayed on the machine code of the Intel x86 architecture. (The TAL researchers display a variant of the familiar 'Intel inside' logo modified to say 'Types inside', and their motto is 'What do you want to type today?'.) TAL was developed by Greg Morrisett and others at Cornell in 1998.

TAL is a refinement of TIL, or Typed Intermediate Language. The TIL project exploited types as the basis for constructing an optimizing compiler for a functional language. The typed functional language is translated to a lower-level typed intermediate language, which is optimized and then compiled to machine code (or, later, TAL). Indeed, most compilers for functional languages now exploit types heavily, in a style similar to that used for TIL. TIL was developed by Greg Morrisett and Bob Harper at Carnegie-Mellon in 1995. Both TAL and TIL take their inspiration from the Girard-Reynolds type system mentioned earlier, John Reynolds also being on the faculty at Carnegie-Mellon.

Finally there is PCC, or Proof Carrying Code. In TAL, as in all type systems, the types are in effect a proof that the program satisfies certain properties. In PCC, instead of adding types to the machine code one adds the proofs directly. The disadvantage of this is that proofs are typically larger than types, and proof checking is slightly harder than type checking; the advantage is that proofs can be more flexible than types. PCC was developed by George Necula and Peter Lee at Carnegie-Mellon in 1996 (Necula moved to Berkeley shortly thereafter).

The first application of PCC was to packet filters. A packet filter needs to be fast (servers process a lot of packets) and trusted (it is part of the kernel). A special-purpose solution is the Berkely Packet Filter, or BPF, a small interpreted language for packet filters built-in to the kernel of Berkeley Unix which enforces the safety policy by run-time checks. One can avoid the overhead of interpretation by using SFI (which adds its own run-time checks) or PCC (where the proof is checked just once, and there is no run-time overhead). The original PCC paper presented measurements showing that SFI was four to six times faster than BPF, and that SCC was another 25% faster that SFI. A drawback of PCC is that it takes a while to check the proof, so SFI is faster than PCC if checking a tiny number of packets, but PCC is faster than SFI if checking more than a dozen packet (the common case). This work won the best paper award at OSDI 96, the Symposium on Operating System Design and Implementation.

Another application of PCC is to our old friend, Java. Instead of transmitting byte codes and using a just-in-time compiler, with the PCC Special J system one can transmit already compiled machine code together with a proof that the code satisfies all the properties that would normally be checked by the byte code verifier or at run-time. Some Java users resort to the native code interface, which gives no security guarantees but provides a way to execute machine code directly; with Special J one can execute machine code directly while still preserving the security guarantees. Necula and Lee have founded a start-up, Cedilla Systems, to pursue commercial possibilities for PCC, and PCC is now patent pending.

One advantage of both TAL and PCC is that the *trusted computing base* is smaller. Designing a secure system is hard — a number of bugs have been spotted and corrected in the Java platform since it was first released. If a platform for secure computing is simple and small, then it is less likely to contain errors. In TAL, the only code that needs to be trusted is the type checker for assembly language, and in PCC the only code that needs to be trusted is that of the type checker. In particular, the compiler that generates the code does *not* need to be trusted. If the compiler contains an error, then it is likely to generate code that does not type check (for TAL), or an incorrect proof (for PCC). Thus, TAL and PCC, as well as enhancing security, also aid in debugging the compiler.

In rare cases, it might be that the compiler generates code that does not do what was expected, but still typechecks or has a correct proof. Even in this case, the types or proof will guarantee that the code cannot violate the security policy. For instance, the security policy might ensure that the code never indexes outside the bounds of a given array, but the code might still look at the wrong array index.

What happens if a malicious user changes the code? Then the type or proof will not match the code, and the program will not be run. What happens if a malicious user changes the code, and changes the types or proofs to match the new code? Then the types or proofs still guarantee that the new code cannot violate the security policy, even though the code may not behave as expected. (There is not much difference here between a malicious user and an erroneous compiler!) The flip side of this is that neither TAL nor PCC is a panacea. One needs to set the security policy in advance, and design the type or proof system to enforce that policy.

TAL and PCC have attracted a great deal of interest in the research community — there is a biannual workshop devoted to typed compilers in the style of TIL, and there is ongoing research on PCC at Princeton, Purdue, and Yale, as well as Berkeley and Carnegie Mellon.

Both TAL and PCC work in much the same way. They were both designed by researchers in functional languages, and they depend heavily on the logics and type systems whose roots were traced in this paper. As it happens, they were also initiated by researchers working in the same place, Carnegie Mellon, at the same time, just a few years ago. The similarities were recognized from the start, and research on each has reinforced the other.

By now this should be almost completely unsurprising, since as we've seen types and proofs are strikingly similar, and, in the right circumstances, can even be in precise correspondence. With Gentzen and Curry it took thirty years for the underlying similarities to be spotted, with Girard and Reynolds the same idea popped up at nearly the same time for very different reasons. So the only surprise with

TAL and PCC is that the similarities were recognized from the start!

It is too early to say, but just as physics is underpinned by the work of Copernicus, Galileo, and Newton, one day computing may be underpinned by the work of Frege, Gentzen, and Church.