Proof assistants in computer science research

Xavier Leroy

Inria Paris-Rocquencourt

Semantics of proofs and certified mathematics, 2014-04-22
Mathematical models and proofs are essential in many areas of C.S., for validation as well as for discovery.
A tension

Mathematics → Engineering

Elegant, uncluttered abstractions → Practical, overly detailed artifacts
A vicious circle

"I have an idea!"

Simple formal system

Happy reviewers

Decent proofs

"Let's make it more realistic!"

Big, complicated system

Excruciatingly long / fast and loose proofs

Exhausted reviewers
A vicious circle

“I have an idea!”

“Let’s make it more realistic!”

Simple formal system

Big, complicated system

Happy reviewers

Exhausted reviewers

Decent proofs

Excruciatingly long / fast and loose proofs
Machine assistance to the rescue?

Proofs written by computer scientists are boring: they read as if the author is programming the reader.

(John C. Mitchell)

Who said that the reader must be human? Proofs can and should be checked by computers.

(The proof assistant community)
In this talk

Three short stories where the use of proof assistants enables C.S. research to scale properly:

- Programming languages metatheory (POPLmark)
- Deductive verification of critical software (seL4)
- Formally-verified compilation (CompCert)
Part I

The metatheory of programming languages
Formally defining a programming language

Syntax
(what do programs look like?)

Dynamic semantics
(how programs execute? what do they compute?)

Type system / Static semantics
(what are well-formed programs?)
A trivial language: simply-typed \( \lambda \)-calculus

Syntax:  \( \text{Expr} \ni a ::= N \mid x \mid \lambda x.a. \mid a_1 \ a_2 \)

Dynamic semantics:  \((\lambda x.a_1) \ a_2 \rightarrow a_1[x \leftarrow a_2]\)

Type system:

\[
\begin{align*}
\Gamma \vdash N : \text{nat} & \quad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \Gamma(x)} & \quad \frac{\Gamma, \ x : \tau \vdash a : \tau'}{\Gamma \vdash \lambda x.a : \tau \rightarrow \tau'} \\
& \quad \frac{\Gamma \vdash a_1 : \tau \rightarrow \tau' \quad \Gamma \vdash a_2 : a_2 : \tau}{\Gamma \vdash a_1 \ a_2 : \tau'}
\end{align*}
\]
The metatheory of a programming language

A pretentious word to refer to important properties that hold for all well-typed programs, e.g.

- **Type soundness**: execution never crashes on an undefined computation
  \[ \forall a. \ a \to \cdots \not\rightarrow \mathbb{N} \ b \]

- **Normalization**: execution always terminates
  \[ \forall a, \exists b, \ a \to \cdots \to b \not\rightarrow \]

Plus: decidability of type-checking; existence of principal types; etc.
Scaling up: System Fsub

Adds support for **polymorphism** and **subtyping** as in OO languages.

\[
\begin{align*}
(x : \tau) &\in \Gamma \\
\Gamma &\vdash x : \Gamma(x) \\
\Gamma &\vdash a_1 : \tau \to \tau' \quad \Gamma &\vdash a_2 : a_2 : \tau \\
\Gamma &\vdash a_1 \ a_2 : \tau'
\end{align*}
\]

\[
\Gamma, \ x : \tau \vdash a : \tau'
\]

\[
\Gamma \vdash \lambda x. a : \tau \to \tau'
\]

\[
\Gamma \vdash a : \tau \quad \Gamma \vdash \tau <: \tau'
\]

\[
\Gamma \vdash a : \tau'
\]

\[
\Gamma, \ X <: \tau \vdash a : \tau'
\]

\[
\Gamma \vdash \forall X <: \tau. a : \forall X <: \tau. \tau'
\]

\[
\Gamma \vdash a : \forall X <: \tau_1. \tau_2 \quad \Gamma \vdash \tau <: \tau_1
\]

\[
\Gamma \vdash a[\tau] : \tau_2[X \leftarrow \tau]
\]
Scaling up: System Fsub

Subtyping rules:

\begin{align*}
\Gamma \vdash \tau <: \top & \quad \Gamma \vdash \mathcal{X} <: \mathcal{X} \\
\Gamma \vdash \tau_1' <: \tau_1 & \quad \Gamma \vdash \tau_2 <: \tau_2' \\
\hline
\Gamma \vdash \tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2' \\
\hline
\Gamma \vdash \forall \mathcal{X} <: \tau_1. \tau_2 <: \forall \mathcal{X} <: \tau_1'. \tau_2' \\
(X <: \tau') \in \Gamma & \quad \Gamma \vdash \tau' <: \tau \\
\hline
\Gamma \vdash \mathcal{X} <: \tau \\
\end{align*}
Kernel Fsub: type-checking is decidable.

\[
\Gamma, X <: \tau \vdash \tau_2 <: \tau'_2
\]

\[
\Gamma \vdash \forall X <: \tau. \tau_2 <: \forall X <: \tau. \tau'_2
\]

Full Fsub: type-checking is undecidable.

\[
\Gamma \vdash \tau'_1 <: \tau_1 \quad \Gamma, X <: \tau_1 \vdash \tau_2 <: \tau'_2
\]

\[
\Gamma \vdash \forall X <: \tau_1. \tau_2 <: \forall X <: \tau'_1 \cdot \tau'_2
\]
Growing pains

So many type systems to explore!

- Type more features: imperative, object-orientation, concurrency, distribution, . . .
- Type them more precisely: polymorphism, type abstraction, refinement types, dependent types, . . .
- Extend type safety to low-level languages: virtual machines, assembly.

Metatheory proofs become intractable:

- Large case analyses (e.g. 20-page appendix).
- Interesting cases are lost in a sea of routine cases.
- The patience of reviewers is exhausted.
The Grail: formalizing real-world programming languages

Book-sized specifications; hundreds of inference rules.
Very little can be proved about them.
A vision:

*How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?*

A challenge: mechanize the metatheory of Fsub in the proof assistant of your choice.

The result: 15 solutions (+/- complete), in 7 proof assistants.
The good news

All solutions handle part 1 of the challenge, which is the most difficult proof of Fsub’s metatheory.

**Theorem (Transitivity of subtyping)**

>If \( \Gamma \vdash \tau_1 <: \tau_2 \) and \( \Gamma \vdash \tau_2 <: \tau_3 \) then \( \Gamma \vdash \tau_1 <: \tau_3 \).

Proof: induction on the size of \( \tau_2 \) and mutual induction with

**Theorem (Narrowing)**

>If \( \Gamma, X <: \tau_1, \Delta \vdash \tau <: \tau' \) and \( \Gamma \vdash \tau_2 <: \tau_3 \), then \( \Gamma, X <: \tau_2, \Delta \vdash \tau <: \tau' \).

Moreover, the mechanized proofs are not much bigger than a detailed pencil-and-paper proof.
The bad news

All solutions have a hard time dealing with variable bindings and invariance by renaming of bound variables ($\alpha$-equivalence).

$$\forall X. P(X)$$

$$\sum_{i=0}^{i=n} i^2$$

$$\int_0^{\infty} f(x)\, dx$$

$$x \mapsto e^{-x}$$
The bad news

All solutions have a hard time dealing with variable bindings and invariance by renaming of bound variables ($\alpha$-equivalence).

\[
\forall X. \ P(X) \ = \ \forall Y. \ P(Y)
\]

\[
\sum_{i=0}^{i=n} i^2 \ = \ \sum_{j=0}^{j=n} j^2
\]

\[
\int_0^{\infty} f(x) \, dx \ = \ \int_0^{\infty} f(t) \, dt
\]

\[
x \mapsto e^{-x} \ = \ z \mapsto e^{-z}
\]

\[
\ne \mapsto e^{-e}
\]

name capture!
Approaches to variable bindings

Either: special support in the logic and proof assistant:
- Nominal logic (Pitts et al);

Or: special encodings so that $\alpha$-equivalence is equality:
- de Bruijn indices;
- locally nameless techniques;
- parametric HOAS;
- etc.

No “best” approach yet.

Definitions and statements do not look exactly like on paper.
Several successes for practically-important languages:

- Java and the JVM (Klein, Lochbihler, Nipkow)
- Standard ML (Crary, Harper)
- C (Norrish, Leroy, Krebbers)
- Javascript (Gardner et al)
- x86 machine language (Morrisett et al, Myreen, Benton)

These formalizations are usable (and used) to prove specific programs as well as general metatheoretic results.

15% of POPL submissions now include a machine-checked proof.
Part II

Formal verification of critical software
The software reliability landscape

Impact of bugs

- Frustration
- Loss of time
- Frequent upgrades

Software kind

- Ordinary
- PC software
- Smartphones
- ERP

Sensitive
- Loss of money
- Bad PR
- Getting sued

Data
- Security
- Network security
- Privacy on Internet

Critical
- Someone dies
- All of the below

Medical
- Railways
- Nuclear plants
- Airplanes
The software reliability landscape

Impact of bugs

- Frustration
- Loss of time
- Frequent upgrades

Software kind

- Ordinary:
  - Data security
  - Network security
  - Privacy on Internet
- Sensitive:
  - Loss of money
  - Bad PR
  - Getting sued
- Critical:
  - Medical
  - Railways
  - Nuclear plants
  - Airplanes
The software reliability landscape

Impact of bugs

- Someone dies
- All of the below
- Loss of money
- Bad PR
- Getting sued
- Frustration
- Loss of time
- Frequent upgrades

Software kind

- Critical
- Sensitive
- Ordinary

- Railways
- Medical
- Nuclear plants
- Airplanes
- Data security
- Network security
- Privacy on Internet
- PC software
- Smartphones
- ERP
Validating critical software

*Program testing can be used to show the presence of bugs, but never to show their absence!*

(E.W.Dijkstra, 1972)

Dominant approach: testing.

Alternative on the rise: tool-assisted formal verification: verify, possibly infer, properties that hold of all possible executions of a program.

Used in some industrial contexts (airplanes, railways)

- To obtain independent guarantees (besides testing).
- To obtain stronger guarantees (than with testing).
- To save time and money (rigorous testing is expensive).
A panorama of verification tools

- Basic safety
- Full correctness

Specialized logics
- F.O. logic
- H.O. logic + induction

Automatic

Interactive

LOC

- Static analyzers
- Model checkers
- Deductive program provers
- Proof assistants
Example: computing prime numbers
Knuth, *The Art of Computer Programming*, vol. 1

```
int a[] = new int[n];
a[0] = 2;
loop:
  for (int i = 1, m = 3; i < n; m = m + 2) {
    int j = 0;
    while (j < i ∧ a[j] <= \sqrt{m} ) {
      if (a[j] divides m) continue loop;
      j = j + 1;
    }
    a[i] = m; i = i + 1;
  }
```

**Goal:** compute the first $n$ prime numbers.

**Algorithm:** try successive odd numbers $m$, striking out those divisible by primes already found.
Example: computing prime numbers

Knuth, *The Art of Computer Programming*, vol.1

```java
int a[] = new int[n];
a[0] = 2;
loop:
    for (int i = 1, m = 3; i < n; m = m + 2) {
        int j = 0;
        while (j < i ∧ a[j] <= √m ) {
            if (a[j] divides m) continue loop;
            j = j + 1;
        }
        a[i] = m; i = i + 1;
    }
```

**Static analyzer:** can infer $1 \leq i < n$ and $0 \leq j < i$ inside the loop, hence array accesses are safe (within bounds).
Example: computing prime numbers

Knuth, *The Art of Computer Programming*, vol.1

```java
int a[] = new int[n];
a[0] = 2;
loop:
    for (int i = 1, m = 3; i < n; m = m + 2) {
        int j = 0;
        while (j < i ∧ a[j] <= √m) {
            if (a[j] divides m) continue loop;
            j = j + 1;
        }
        a[i] = m; i = i + 1;
    }
```

**Automatic program prover**: can prove partial correctness if the user provides detailed loop invariants and simple axioms about primality and divisibility. (Termination is harder to prove.)
Example: computing prime numbers
Knuth, *The Art of Computer Programming*, vol.1

```java
int a[] = new int[n];
a[0] = 2;

loop:
  for (int i = 1, m = 3; i < n; m = m + 2) {
    /* invariant:
     \( \forall k, 0 \leq k < i \Rightarrow \text{isprime}(a[k]) \)
     \( \forall p, 2 \leq p < m \land \text{isprime}(p) \Rightarrow \exists k, 0 \leq k < i \land a[k] = p \)
     \( \forall k, m, 0 \leq k < j < i \Rightarrow a[k] < a[j] \)
    */
  }
```

**Automatic program prover:** can prove partial correctness if the user provides detailed loop invariants and simple axioms about primality and divisibility. (Termination is harder to prove.)
Example: computing prime numbers

Knuth, *The Art of Computer Programming*, vol.1

```java
int a[] = new int[n];
a[0] = 2;

loop:
    for (int i = 1, m = 3; i < n; m = m + 2) {
        int j = 0;
        while (j < i ∧ a[j] <= √m) {
            if (a[j] divides m) continue loop;
            j = j + 1;
        }
    }
...
```

Knuth’s cunning optimization: the test $j < i$ is redundant and can be omitted. Can you see why?
Example: computing prime numbers
Knuth, *The Art of Computer Programming*, vol.1

```java
int a[] = new int[n];
a[0] = 2;
loop:
    for (int i = 1, m = 3; i < n; m = m + 2) {
        int j = 0;
        while (j < i ∧ a[j] <= \sqrt{m}) {
            if (a[j] divides m) continue loop;
            j = j + 1;
        }
    }
...
```

Knuth’s cunning optimization: the test $j < i$ is redundant and can be omitted. Can you see why? Because of Bertrand’s postulate!

**Theorem (Chebychev)**

*For all $n \geq 1$, there exists a prime $p$ in $[n, 2n]$.*

(Coq proof: Laurent Théry, 2002.)
Scaling up: the seL4 verified microkernel
(G. Klein et al, NICTA)

The security core of an operating system.

<table>
<thead>
<tr>
<th>Traditional approach</th>
<th>Microkernel approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>App 1</td>
<td>App 1</td>
</tr>
<tr>
<td>App 2</td>
<td>App 2</td>
</tr>
<tr>
<td></td>
<td>protected</td>
</tr>
<tr>
<td>monolithic kernel</td>
<td>OS services</td>
</tr>
<tr>
<td>(10^7 LOC)</td>
<td>OS services</td>
</tr>
<tr>
<td>hardware</td>
<td>micro-kernel</td>
</tr>
<tr>
<td></td>
<td>(10^4 LOC)</td>
</tr>
<tr>
<td></td>
<td>hardware</td>
</tr>
</tbody>
</table>
Verifying seL4 with Isabelle/HOL
(G. Klein et al, NICTA; ACM TOPLAS 32(1), 2014)
Verifying seL4 with Isabelle/HOL
(G. Klein et al, NICTA; ACM TOPLAS 32(1), 2014)

- Haskell prototype
- Optimized C implementation
- High-level abstract specs
- Automatic generation
- Refinement proof (semi-automatic)
- Refinement proof (interactive)
- Generic security properties
- Semantics of C implementation
Verifying seL4 with Isabelle/HOL
(G. Klein et al, NICTA; ACM TOPLAS 32(1), 2014)
Verifying seL4 with Isabelle/HOL

(G. Klein et al, NICTA; ACM TOPLAS 32(1), 2014)
A dream comes true.
(≈ 15 unsuccessful OS verification projects since the late 1970’s).

No compromise on the performance of the OS.
(≈ 80% of the speed of the fastest unverified L4-style kernel.)

The largest deductive verification of a software system ever:
20 person.year, 200+ KLOC proofs.
Part III

Formally-verified compilation
Trust in software verification

- Simulation
- Model-checking
- Program proof
- Static analysis
- Testing

Simulink, Scade → Code generator → C code → Compiler → Executable

The unsoundness risk: Are verification tools semantically sound?
The miscompilation risk: Are compilers semantics-preserving?
NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.

http://www.nullstone.com/htmls/category/divide.htm

We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables.

E. Eide & J. Regehr, EMSOFT 2008

To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input.

X. Yang, Y. Chen, E. Eide & J. Regehr, PLDI 2011
An example of optimizing compilation

double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}

Compiled with a good compiler, then manually decompiled back to C...
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
    L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
    L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
    L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
    L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
    L5: return dp;
    L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
double dotproduct(int n, double a[], double b[])
{
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
L5: return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
L5: return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
Addressing miscompilation

Best industrial practices: more testing; manual reviews of generated assembly code; turn optimizations off; . . .

A more radical solution: why not formally verify the compiler itself?

After all, compilers have simple specifications:

*If compilation succeeds, the generated code should behave as prescribed by the semantics of the source program.*

As a corollary, we obtain:

*Any safety property of the observable behavior of the source program carries over to the generated executable code.*
1. Introduction. This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.
Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch
Computer Science Department
Stanford University

Abstract
We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

Machine Intelligence (7), 1972.
The CompCert project
(X.Leroy, S.Blazy, et al)

Develop and prove correct a realistic compiler, usable for critical embedded software.

- Source language: a very large subset of C99.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  ⇒ careful code generation; some optimizations.

Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler.
The formally verified part of the compiler

- **CompCert C**: side-effects out of expressions
- **Clight**: type elimination loop simplifications
- **C#minor**: stack allocation of “&” variables
- **RTL**: CFG construction, expr. decomp.
- **CminorSel**: instruction selection
- **LTL**: register allocation (IRC), calling conventions
- **Linear**: linearization of the CFG
- **Mach**: layout of stack frames, asm code generation
- Asm x86
- Asm ARM
- Asm PPC

Optimizations: constant prop., CSE, inlining, tail calls
Formally verified using Coq

The correctness proof (semantic preservation) for the compiler is entirely machine-checked, using the Coq proof assistant.

Theorem transf_c_program_preservation:
  forall p tp beh,
  transf_c_program p = OK tp ->
  program_behaves (Asm.semantics tp) beh ->
  exists beh’, program_behaves (Csem.semantics p) beh’ /\ behavior_improves beh’ beh.
Compiler verification patterns (for each pass)

Verified transformation

External solver with verified validation
100,000 lines of Coq.

Including 15000 lines of “source code” (≈ 60,000 lines of Java).

6 person.years

Low proof automation (could be improved).
Programmed (mostly) in Coq

All the verified parts of the compiler are programmed directly in Coq’s specification language, using pure functional style.

- Monads to handle errors and mutable state.
- Purely functional data structures.

Coq’s extraction mechanism produces executable Caml code from these specifications.

Claim: purely functional programming is the shortest path to writing and proving a program.
The whole CompCert compiler

- C source → AST C
  - parsing, construction of an AST
  - type-checking, de-sugaring

- AST C → AST Asm
  - Register allocation
  - Code linearization heuristics

- AST Asm → Assembly
  - printing of asm syntax

- Assembly → Executable
  - assembling
  - linking

Part of the TCB
- Not proved (hand-written in Caml)
- Proved in Coq (extracted to Caml)

Not part of the TCB
Performance of generated code
(On a Power 7 processor)
The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

X. Yang, Y. Chen, E. Eide, J. Regehr, PLDI 2011
Part IV

Conclusions
In closing. . .

Proof assistants enable unprecedented scaling in many areas of computer science:

- in size and realism of the formal systems considered;
- in mathematical assurance.

Additional benefits:

- Make research papers easier to write and to read.
- Give a second chance to students/engineers/scientists who are insecure in their abilities to do mathematics on paper.
- Facilitate collaborative work of the free software kind.
Some points to keep in mind

Mechanized proofs do not eliminate errors, they reduce the errors to the definitions and statements of theorems.

Proof assistants are addictive and a huge time sink.

Proper engineering of specifications and proofs is crucial.

Fragmentation of the community around multiple theorem provers. (Just like programming languages.)

Mechanized proofs require maintenance and proper archival.
Go forth and mechanize!

For more information on the projects presented:
POPLmark: http://www.seas.upenn.edu/~plclub/poplmark
CompCert: http://compcert.inria.fr/