Machine-checked proofs of program correctness

COS 326
Andrew W. Appel
Princeton University

In this course, you saw

- how to prove that functional programs are correct

```
let easy x y z = x * (y + z)

Theorem:  for all integers n, m, k, easy k n m == easy k m n
Proof:
     easy k n m                    (left-hand side of equation)
== k * (n + m)                     (by def of easy)
== k * (m + n)                     (by math)
== easy k m n                      (by def of easy)
QED.
```

- But . . .

Programs are *software*;   software is *big*; software keeps *changing* as it is maintained;

How do you keep the proof (in github?) sync'ed up with the software in github ?

The answer

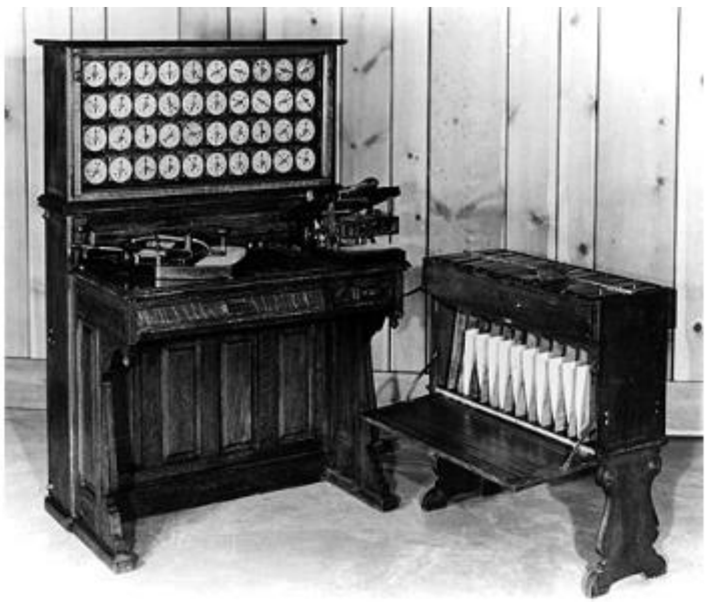
- Proofs are text files in a formal language, just like software
- Check the correctness of proofs by computer

- In fact, it’s only a slight exaggeration to say,

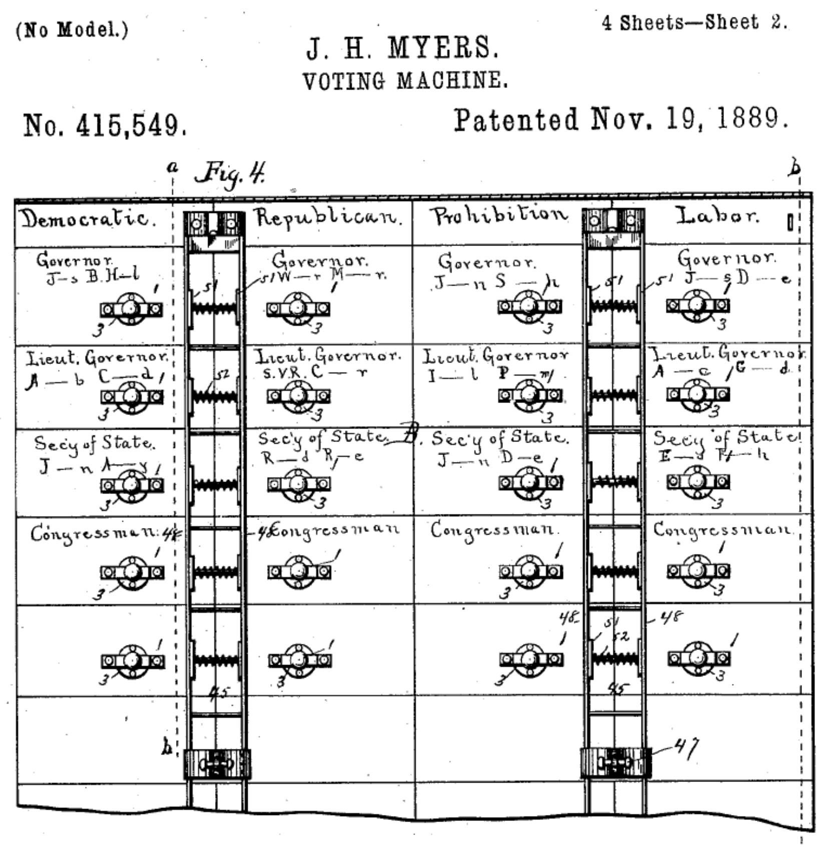
“Checking proofs is what computers were invented for!”

Back around 1890 . . .

Herman Hollerith developed punch-card sorters to tabulate the 1890 census

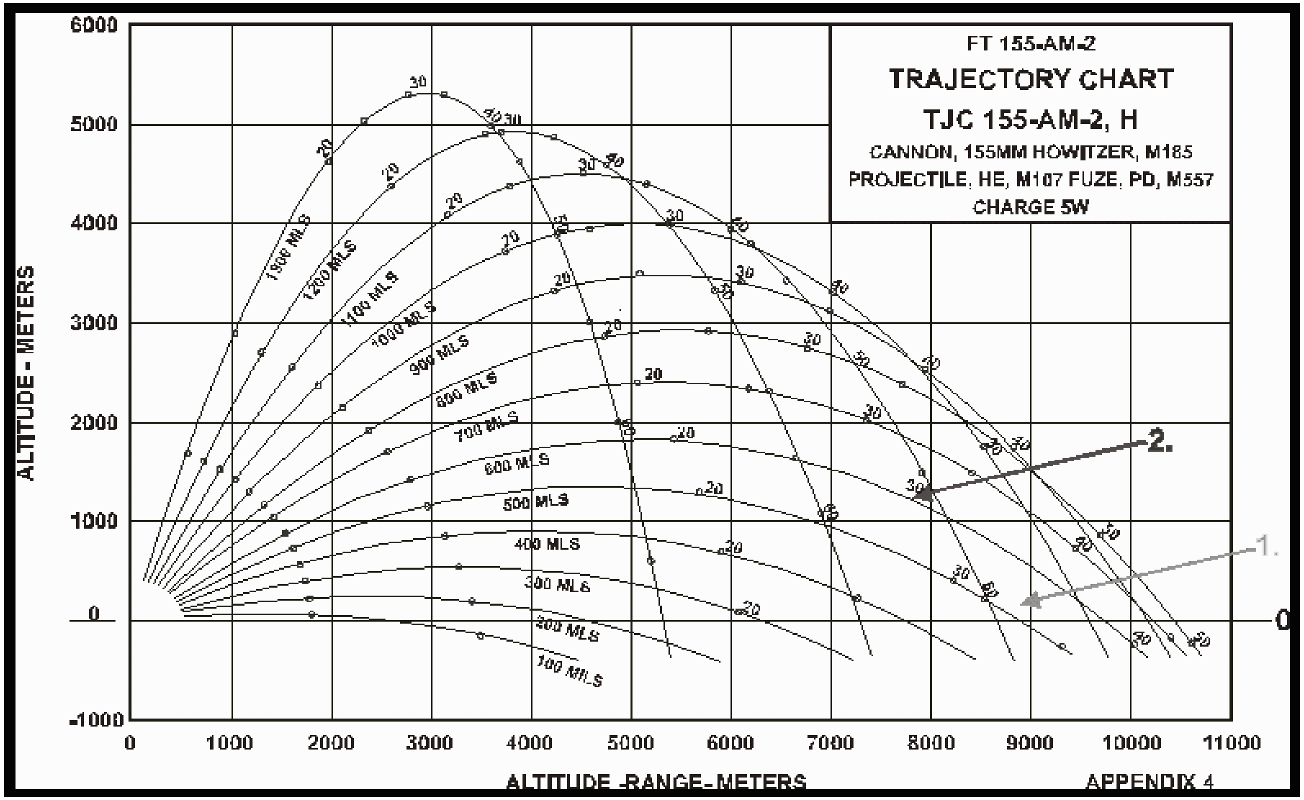


Others invented mechanical voting machines, cash registers, counters, adding machines…



Computing

Automated mechanical calculators were much used 1900-1945 to calculate *ballistics tables* for artillery



What computers looked like in 1925

com·put·er
kəmˈpyo͞odər/
noun

1. an electronic device for storing and processing data, typically in binary form, according to instructions given to it in a variable program.

2. a person who makes calculations, especially with a calculating machine.

Hilbert's "program" of 1920

David Hilbert
1862-1943

German mathematician (Most famous mathematician in the world, circa 1900-1925)

In 1920 Hilbert proposed explicitly a research project that became known as *Hilbert's program.* He wanted mathematics to be formulated on a solid and complete logical foundation. He believed that in principle this could be done, by showing that:

1. all of mathematics follows from a correctly chosen finite system of axioms; and

2. that some such axiom system is provably consistent through some means such as the epsilon calculus.

-- Wikipedia

Wittgenstein and Gödel, 1930

Ludwig Wittgenstein,
1889 - 1951

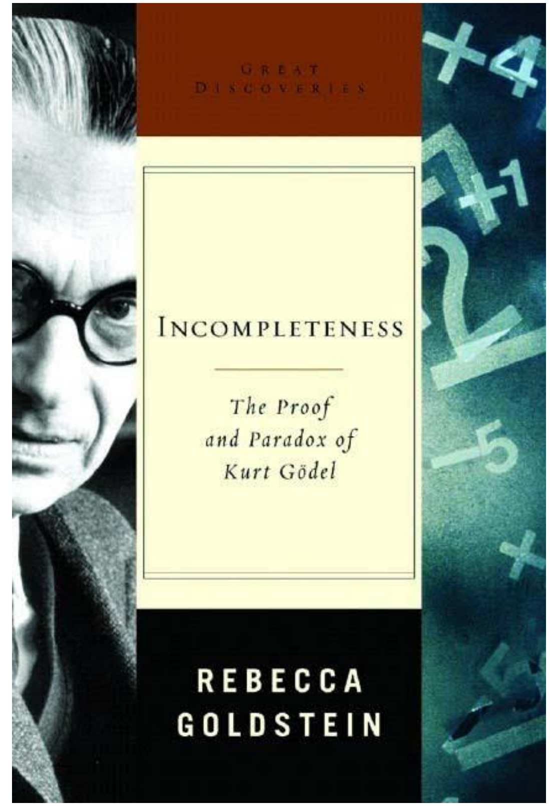
1920s Vienna Circle;

"Logical Positivism" – nothing can be true unless there is a direct way to *observe* it or *prove* it.

Kurt Gödel
1906-1978

Attended meetings of the Vienna Circle; was sure that there are true things that cannot be proved; was desperate to prove Wittgenstein wrong

A good book . . .



Rebecca Goldstein

Princeton PhD 1977 (Philosophy)

Major American author:

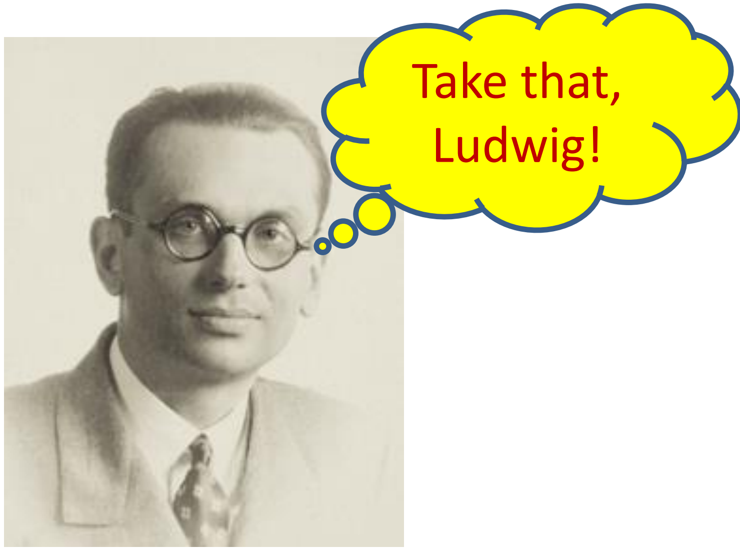
6 important novels

and 3 biographical studies.

Gödel's incompleteness theorem, 1931

In any consistent axiomatization of mathematics that's at least expressive enough to have quantification, addition, and multiplication,

there exist statements that can be neither proved nor disproved (and consequently) there exist true statements that cannot be proved.



Even more important, from our point of view as computer scientists, was <u>*how*</u> Gödel did it. He showed that math formulas can be represented as *numbers*, and syntactic manipulation can be done with *arithmetic.*

That is, he invented ***data structures to represent semantic concepts.***

Hilbert's "program" of 1920

David Hilbert
1862-1943

German mathematician (Most famous mathematician in the world, circa 1900-1925)

In 1920 Hilbert …

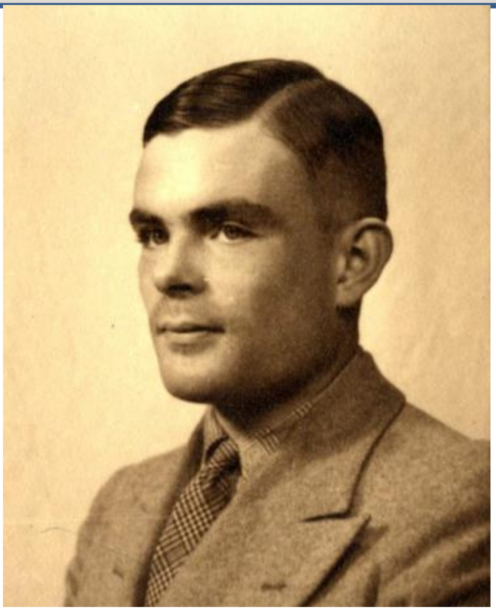
2. th… consistent through … such as the epsilon calculus.

-- Wikipedia

Dang! So much for my project.
Can't find proofs for every *true* statement; can't decide whether every statement is *true.*
But maybe we can salvage something; perhaps a mechanical way *decide* whether a statement is *provable*.

Turing's undecidability theorem, 1936

1936 result: Hilbert's project, a "decision procedure" for mathematical theorems, is impossible.



Alan Turing

1912-1954

English mathematician

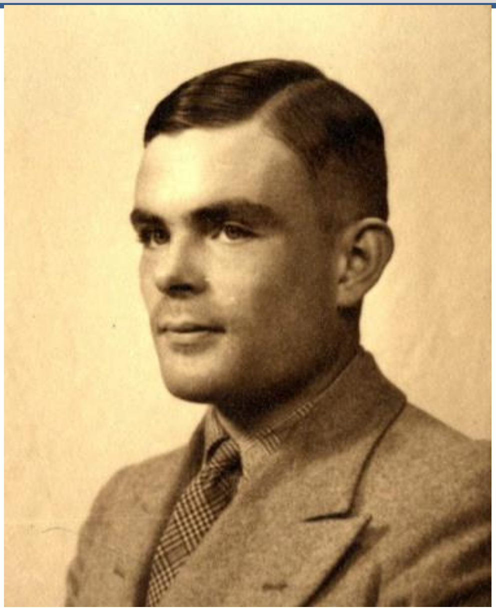
Princeton PhD 1938

Even more important, from our point of view as computer scientists, was <u>*how*</u> Turing did it. His first step in attacking Hilbert's project was that somebody had to formalize the notation of "proof search," and to do this he invented the concept of the *general-purpose computer.*

Turing's undecidability theorem, 1936

1. Thesis: Any model of computation can be represented as a finite-state control reading and writing on an unbounded tape.

2. Theorem: No "computer" thus defined, can implement an algorithm for the *decision problem*, to test whether a mathematical theorem is provable.



Alan Turing
1912-1954
English mathematician
Princeton PhD 1938

Even more important, from our point of view as computer scientists, was *how* Turing did it. His first step in attacking Hilbert's project was that somebody had to formalize the notation of "proof search," and to do this he invented the concept of the *general-purpose computer.*

So therefore . . .

Computers were invented for the purpose of checking proofs of theorems!

All those other "computers" 1890-1948 were not *general-purpose.* To change them from computing ballistics tables to something else, you had to change the wiring.

*Well, these "computers" knew how to do other things too. But not the calculators that they are operating.

So therefore . . .

Computers were invented for the purpose of checking proofs of theorems!

But wait! Didn't Turing prove that checking proofs cannot be done by computer?

All those other "computers" 1890-1948 were not *general-purpose.* To change them from computing ballistics tables to something else,
you had to change the wiring.

No! Gödel and Turing showed how to check proofs by calculation.

What they proved is that *no program that always terminates can test the truth or falsity of every statement.*

But it's easy to check proofs if we give the program, as input, both the theorem and the proof of it.

Proof Assistants

Proof Assistant contains

- A small, trusted kernel that checks proofs (which are just abstract-syntax-tree data structures).
- DSL (domain-specific language) for creating proofs
- Libraries in this DSL of lemmas and proof automation
- An interactive development environment

Don't need to trust these, because the kernel rejects invalid proofs

First "Proof Assistant" was invented by Robin Milner at the University of Edinburgh in 1978.

It had an "Object Language" (OL) for proof terms, and a "Meta Language" (ML) for programming the construction of proofs.

That's where is where ML (and Ocaml) came from.

Coq Proof Assistant

Developed by computer scientists 1989-present at INRIA,
Institut National de Recherche en Informatique et en Automatique,
the French national computer-science research lab.

Contains a functional programming language (Gallina)
with a logic (Calculus of Inductive Constructions) for proving things.

Demo!

Coq

```
Fixpoint length {A} (xs: list A) : nat :=
 match xs with
 | nil => 0
 | x::xs' => 1 + length xs'
 end.

Fixpoint app {A} (xs ys: list A) : list A :=
 match xs with
 | nil => ys
 | x::xs' => x :: app xs' ys
 end.

Theorem app_length: forall {A} (xs ys: list A),
  length (app xs ys) = length xs + length ys.
Proof.
  . . .
Qed.
```

OCaml

```
let rec length (xs: ‘a list) : int  =
 match xs with
 | [ ] -> 0
 | x::xs' -> 1 + length xs'.

let rec app (xs ys: ‘a list) : ‘a list  =
 match xs with
 | nil -> ys
 | x::xs' -> x :: app xs' ys.

???
```

Demo!

Coq

```
Inductive tree (A: Type) : Type :=
| Leaf: tree A
| Node: A -> tree A -> tree A -> tree A.

Fixpoint insert (t: tree Z) (i: Z) :=
 match t with
 | Leaf => Node i Leaf Leaf
 | Node j left right => if Z_lt_dec i j
                 then Node j (insert left i) right
                 else if Z_lt_dec j i
                     then Node j left (insert right i)
                     else Node i left right
 end.

Theorem lookup_correct:
  forall lo hi i t,  is_search_tree lo t hi ->
      (lookup t i = true <-> in_tree t i).
```

OCaml

```
type 'a tree =
    Leaf
  | Node of 'a * 'a tree * 'a tree

let rec insert (t:tree) (i:int) =
  match t with
  | Leaf ->
      Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
     if i < j then
      Node (j, insert left i, right)
     else
      Node (j, left, insert right i)

???
```

What can we verify?

Example:

Operating System

Isolation: one user process cannot interfere with (read,write) another.

Liveness:
A user process won't be starved.

Availability:
User process cannot crash the OS.

Functional correctness:
User process computes same result in virtual memory as it would if it owned the whole machine.

Example:

Compiler

Simulation:
Object program has same observable behavior as the source program.

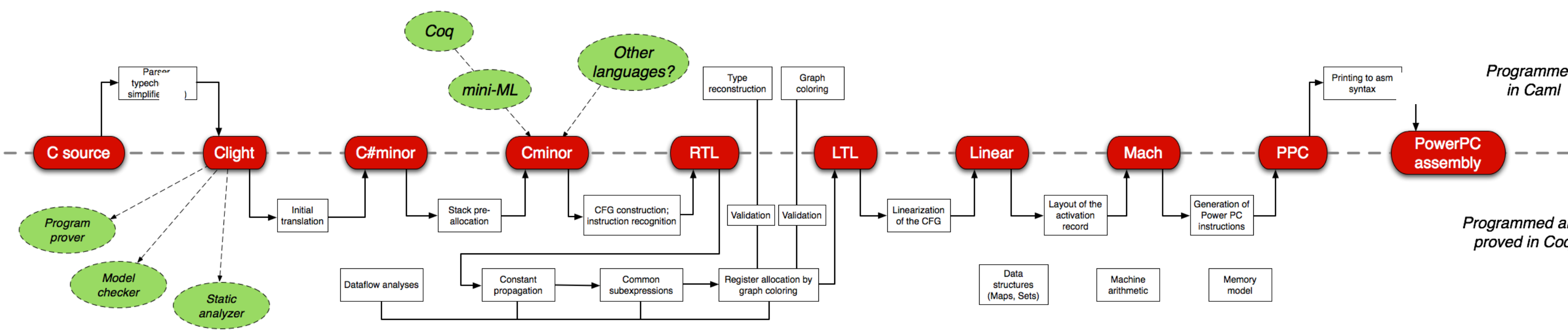
Software verification examples

Application	Relational DBMS *(Harvard)*	Monadic Functional
	Bedrock Web Server *(MIT)*	[other]
	HMAC+SHA *(Princeton)*	C
Compiler	Jinja *(Munich)*	Functional
	CompCert *(INRIA)*	Functional
	VeLLVM *(U. Penn)*	Functional
Operating System	L4.verified *(Australia)*	C
	CertiKOS *(Yale)*	C

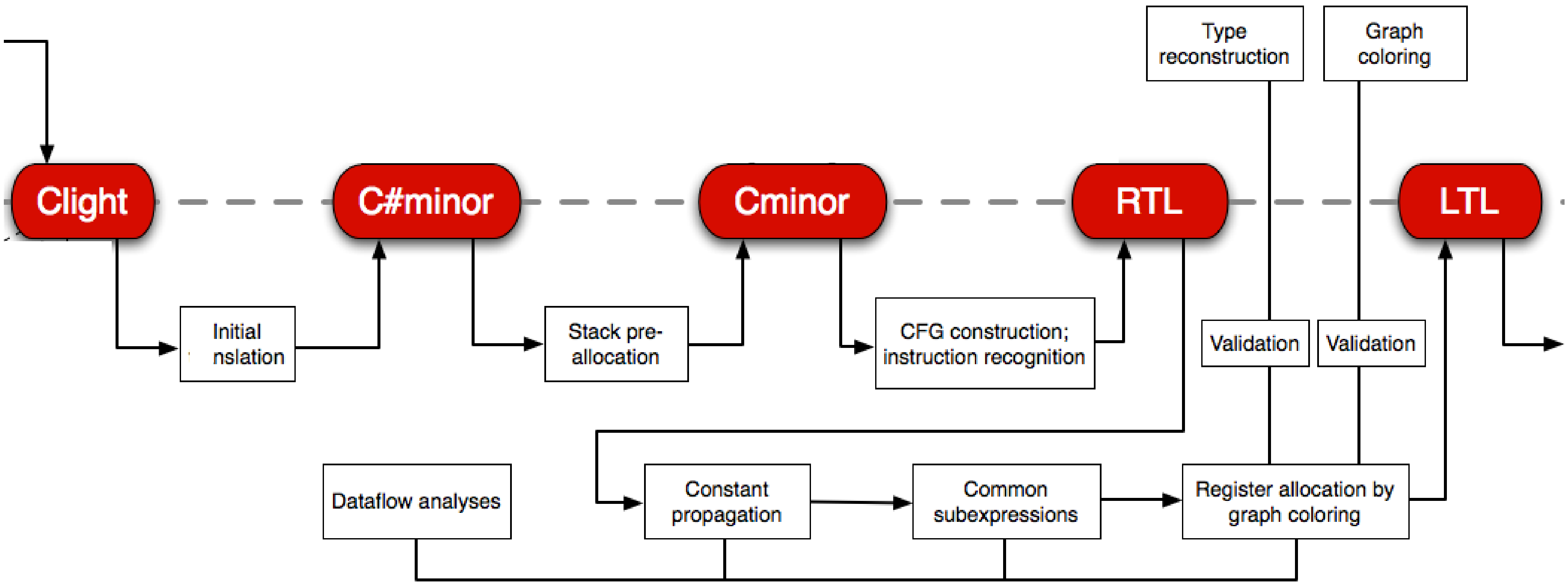
CompCert verified optimizing C compiler

Xavier Leroy
INRIA Rocquencourt

Optimizing C Compiler,
proved correct end-to-end
with machine-checked proof in Coq



CompCert close-up



Verification really works!

Regehr's Csmith project used random testing to assess all popular C compilers, and reported:

``The striking thing about our CompCert results is that the middle-end 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."*

John Regehr
Univ. of Utah

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers." *Proceedings of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*

Conclusion – COS 326 Functional Programming

When I said this semester:

Easier debugging;
fewer embarrassing
security vulnerabilities

Easier reasoning about your code;
supports parallelism well

Program in a safe, functional, programming language with an expressive type system and an efficient implementation.

Type system prevents bugs;
more important, gives you
"invariants for free"

No need to pay a
performance penalty;
map-reduce parallelism is
the *opposite* of a penalty!

Conclusion – COS 326 Functional Programming

And besides,

Program in a safe, functional, programming language with an expressive type system and an efficient implementation.

It's fun!