Programming languages and their trustworthy implementation

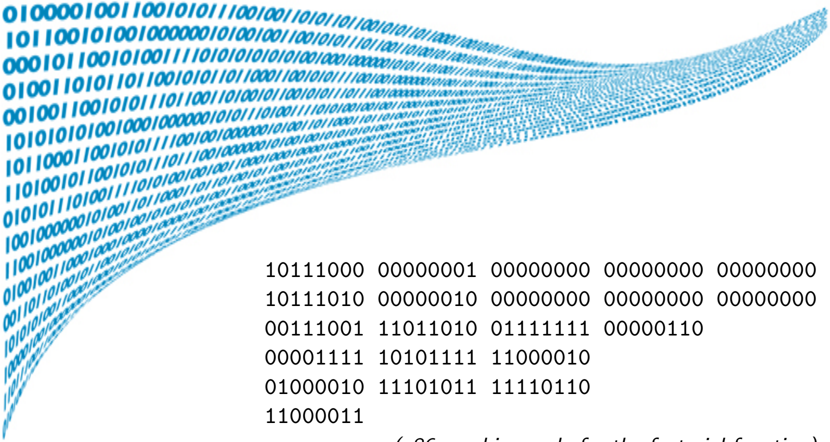
Xavier Leroy

INRIA Paris

Van Wijngaarden award, 2016-11-05

A brief history of programming languages and their compilation

It's all zeros and ones, right?



```
10111000 00000001 00000000 00000000 00000000
10111010 00000010 00000000 00000000 00000000
00111001 11011010 01111111 00000110
00001111 10101111 11000010
01000010 11101011 11110110
11000011
```

(x86 machine code for the factorial function)

Machine code is. That doesn't make it a usable language.

Antiquity (1950): assembly language

A textual representation of machine code, with mnemonic names for instructions, symbolic names for code and data labels, and comments for humans to read.

Example (Factorial in x86 assembly language)

```
; Input: argument N in register EBX
; Output: factorial N in register EAX
Factorial:
        mov eax, 1      ; initial result = 1
        mov edx, 2      ; loop index = 2
L1:     cmp edx, ebx    ; while loop <= N ...
        jg L2
        imul eax, edx   ; multiply result by index
        inc edx         ; increment index
        jmp L1          ; end while
L2:     ret             ; end Factorial function
```

The Renaissance: arithmetic expressions

(FORTRAN 1957)

Express mathematical formulas the way we write them on paper.

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In assembly:

```
mul t1, b, b
mul t2, a, c
mul t2, t2, 4
sub t1, t1, t2
sqrt d, t1
mul t3, a, 2
sub x1, d, b
div x1, x1, t3
neg x2, b
sub x2, x2, d
div x2, x2, t3
```

In FORTRAN:

```
D = SQRT(B*B - 4*A*C)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

A historical parallel with mathematics

Brahmagupta, 628:

> *Whatever is the square-root of the rupas multiplied by the square [and] increased by the square of half the unknown, diminish that by half the unknown [and] divide [the remainder] by its square. [The result is] the unknown.*

Cardano, Viète, et al, 1550–1600:

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The Enlightenment: functions, procedures and recursion

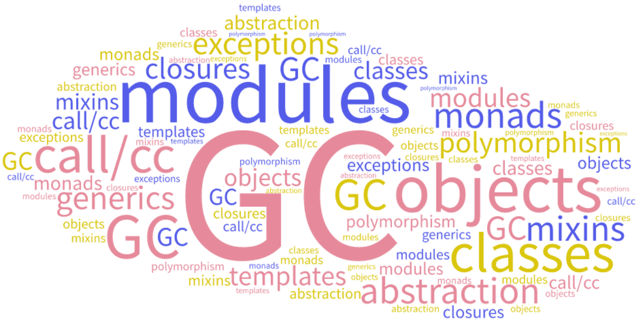
(Lisp, 1958; Algol, 1960)

```
procedure quadratic(x1, x2, a, b, c);
     value a, b, c; real a, b, c, x1, x2;
begin
  real d;
  d := sqrt(b * b - 4 * a * c);
  x1 := (-b + d) / (2 * a);
  x2 := (-b - d) / (2 * a)
end;

integer procedure factorial(n); value n; integer n;
begin
  if n < 2 then
    factorial := 1
  else
    factorial := n * factorial(n-1)
end;
```

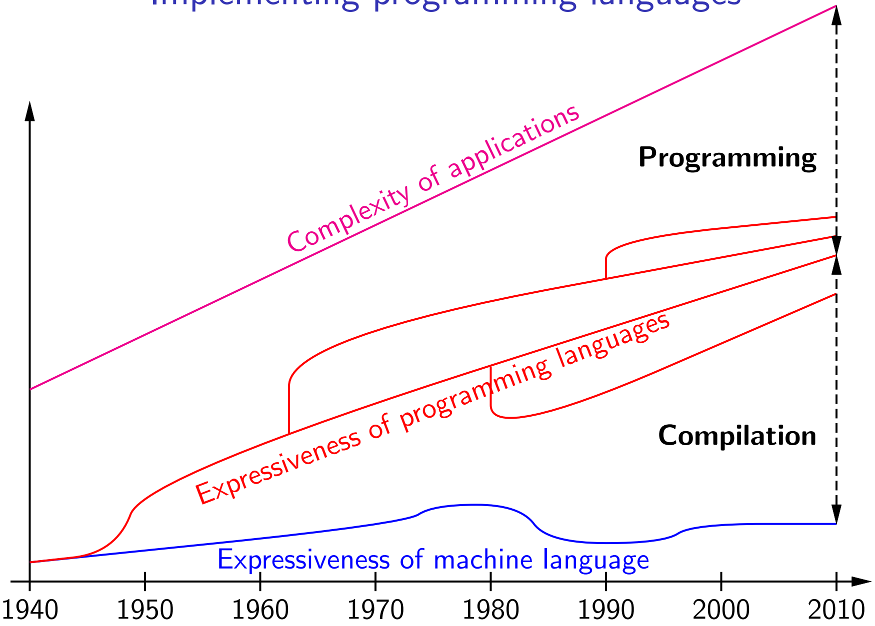
Industrial revolution and modern times

APL 1962, Algol W 1966, ISWIM 1966, BCPL 1967, Algol 1968, Pascal 1970, C 1972, Prolog 1972, ML 1973, CLU 1974, Modula 1975, Smalltalk 1976, Ada 1983, C++ 1983, Common Lisp 1984, Eiffel 1986, Modula-3 1989, Haskell 1990, Python 1991, Java 1995, OCaml 1996, Javascript 1997, C# 2000, Scala 2003, Go 2009, Rust 2010, Swift 2014

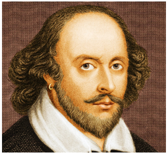


A proliferation of languages that provide support for high-level programming constructs.

Implementing programming languages



The challenge of compilation



1. Translate faithfully a high-level programming language into very low-level machine language.

2. "Optimize", or more exactly improve performance of generated machine code:
   - by taking advantage of hardware features;
   - by eliminating inefficiencies left by the programmer.

An example of optimizing compilation

$$\vec{a} \cdot \vec{b} = \sum_{i=0}^{i<n} a_i b_i$$

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

Compiled with a good compiler, then manually decompiled 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;
}
```

```
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;
```

```
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;









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;
}
```

Can you trust your compiler?

Miscompilation happens

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

Are miscompilation bugs a problem?

For non-critical software:

- Programmers rarely run into them.
- Globally negligible compared with bugs in the program itself.

For critical software:

- A source of concern.
- Require additional verification activities. (E.g. manual reviews of generated assembly code; more tests.)
- Reduce the usefulness of formal verification.
  (A provably-correct source program can still misbehave at run-time!)

Addressing miscompilation

A 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.*

An old idea...

John McCarthy
James Painter[1]

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

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.

Mathematical Aspects of Computer Science, 1967

An old idea...

3

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.

CompCert:

a formally-verified C compiler

The CompCert project

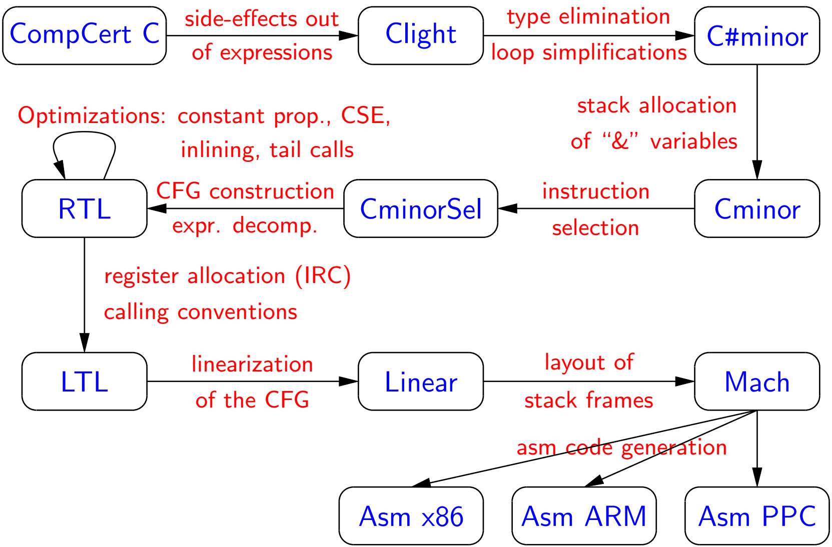
(X. Leroy, S. Blazy, et al)

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

- Source language: a very large subset of C 99.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  $\Rightarrow$ 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



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_correct:
  forall (p: Csyntax.program) (tp: Asm.program)
         (b: behavior),
  transf_c_program p = OK tp ->
  program_behaves (Asm.semantics tp) b ->
  exists b', program_behaves (Csem.semantics p) b'
          /\ behavior_improves b' b.
```

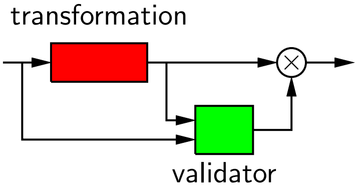
Shows refinement of observable behaviors `beh`:

- Reduction of internal nondeterminism
  (e.g. choose one evaluation order among the several allowed by C)
- Replacement of run-time errors by more defined behaviors
  (e.g. optimize away a division by zero)

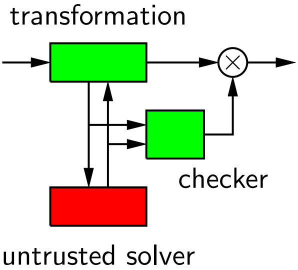
Compiler verification patterns (for each pass)

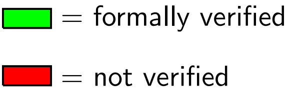
Verified transformation

Verified translation validation



External solver with verified validation





Programmed (mostly) in Coq

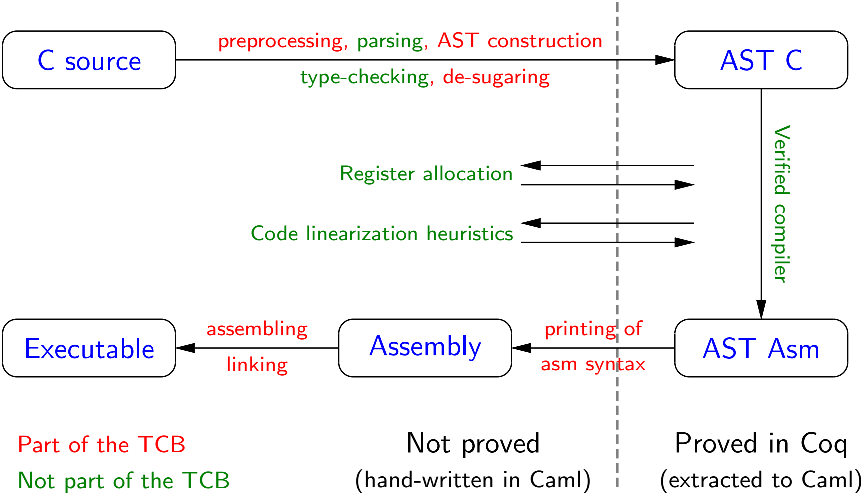
All the verified parts of the compiler are programmed directly in Coq's specification language, using pure functional style.

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

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

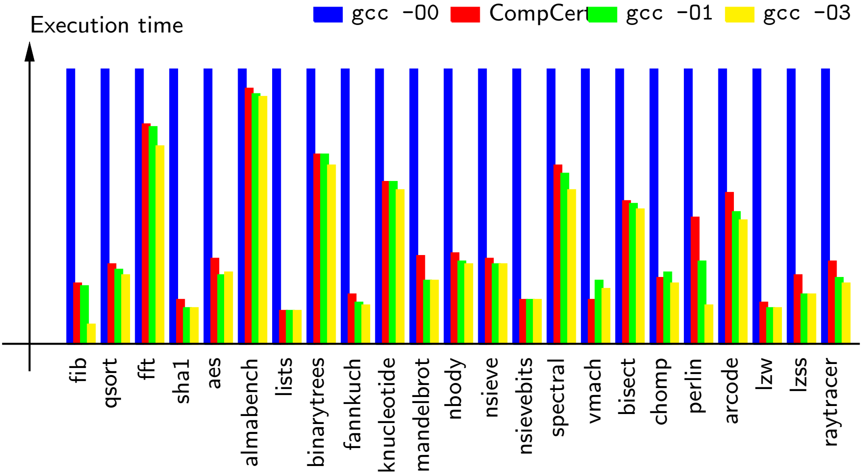
Claim: purely functional programming is the shortest path to writing and proving a program.

The whole Compcert compiler



Performance of generated code

(On a Power 7 processor)



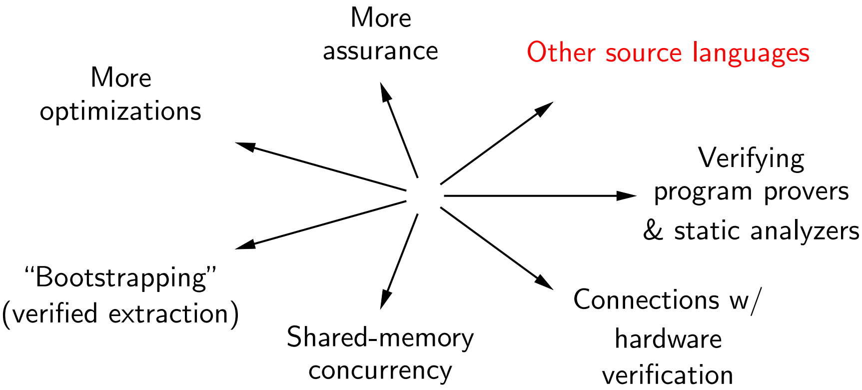
A tangible increase in quality

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

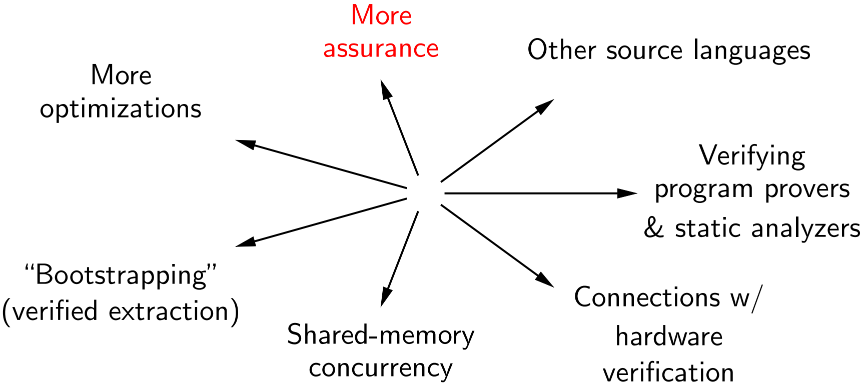
Conclusions and perspectives

Ongoing and future work



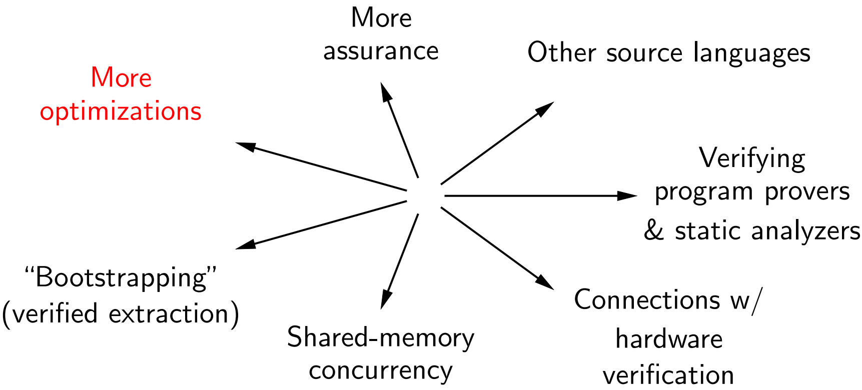
Other source languages besides C: experiments in progress with functional languages, SPARK Ada and SCADE/Lustre.

Ongoing and future work



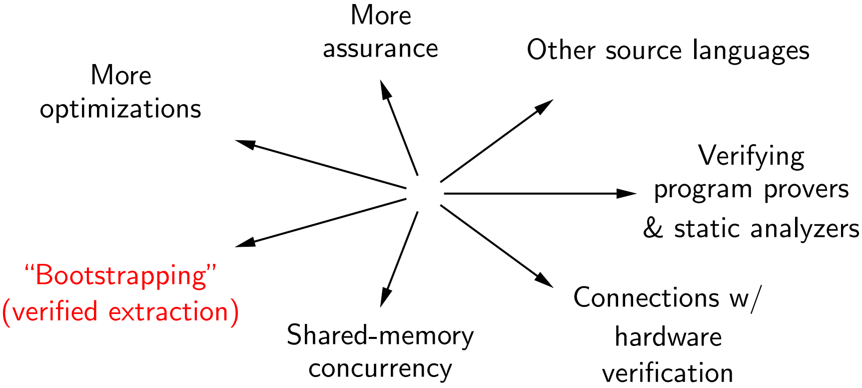
Prove or validate more of the trusted base:
preprocessing, lexing, elaboration, assembling, linking, ...

Ongoing and future work



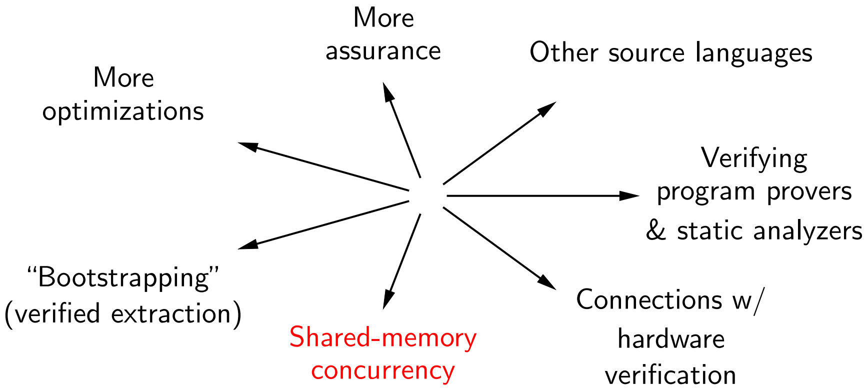
Add advanced optimizations, esp. loop optimizations.

Ongoing and future work



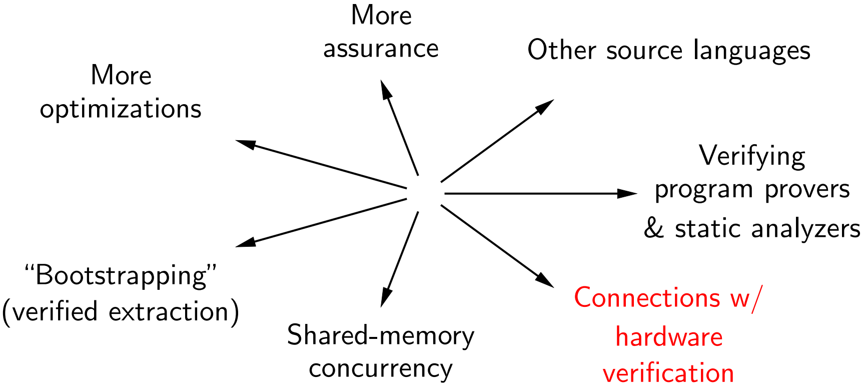
Gain formal confidence in the tools that build CompCert.
(Coq's extraction, OCaml compilation.)

Ongoing and future work



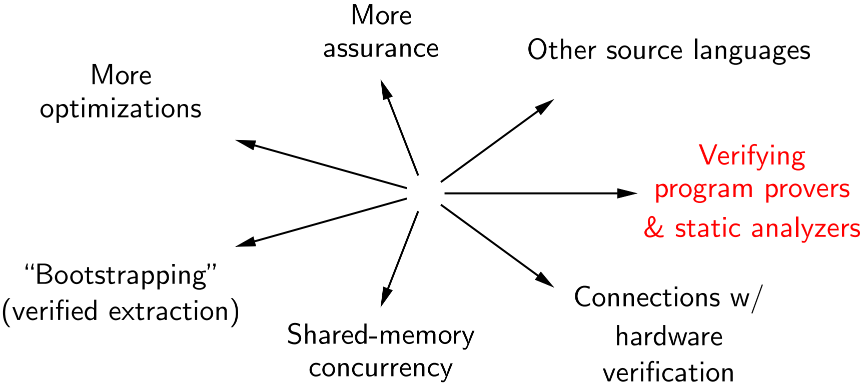
Race-free programs + concurrent separation logic (A. Appel et al)
or: racy programs + hardware memory models (P. Sewell et al).

Ongoing and future work



Formal specs for architectures & instruction sets, as the missing link between compiler verification and hardware verification.

Ongoing and future work



The Verasco project: formal verification of a static analyzer based on abstract interpretation.

In closing...

Critical software deserves the most trustworthy tools
that computer science can produce.

Let's make this a reality!