Ben Greenman
2015-11-19

Abstract

CompCert is a verified C compiler, designed and implemented by Xavier Leroy along with:

- Sandrine Blazy
- Zaynah Dargaye
- Damien Doligez
- Benjamin Grègoire
- Thomas Moniot
- Laurence Rideau
- Bernard Serpette
- Guillarme Claret
- Jacques-Henri Jourdan
- François Pottier
- Silvain Rideau
- Tahina Ramananandro
- Michael Schmidt
- Guillaume Melquiond
- Sylvie Boldo
- Andrew Appel
- Gordon Stewart
- ...and many more

CompCert compiles a safe subset of the C89 and C99 languages. CompCert guarantees that all observable behaviors of compiled code are observable source code behaviors. The project began in 2005 and continues to improve today.

This document is a summary of the CompCert users manual and annotated source code.

1 Real World, Real Compiler

Debugging is always a bummer. This is especially true if buggy software starts losing money, releasing secrets, or killing people. Hence the push for verified software: let's get it right the first time.

Embedded systems is a large application area where mistakes can be fatal. Airplanes, automobiles, construction equipment, spaceships, and the like all run embedded systems. Embedded systems have another constraint: the programs must follow a strict resource limit. For these reasons, the C programming language is commonly used in embedded systems because of its proximity to the hardware and portability.

Debugging embedded systems code is very difficult. The CompCert C compiler offers an efficient, optimizing, and safe C compiler [18]. CompCert claims that if it compiles a program, the executable will only perform only a subset of the actions one would expect the source code to perform. This guarantee does not eliminate debugging—nor does it specify execution time or memory consumption[1]—but it at least rules out translation bugs, thereby encouraging source-program reasoning.

High-assurance software is an old idea. The UK-based Motor Industry Software Reliability Association maintains a specification for safe C programming [2]. Languages like Rust (2009), Go (2007), and Ada (1977) are alternative low-level languages with stronger correctness guaranatees. Companies like Wind River[2] sell high-assurance tools, including the optimizing DIAB compiler toolchain for C. Airbus has been operating *fly-by-wire* planes since the 1987 release of the A320 line; more recently in Boeing released the fly-by-wire 777 in 1994 and the 787 in 2009. These aircraft and their military predecessors were certainly running "verified" software—most likely manually-validated assembly [].

Within the design space of high-assurance code, CompCert is the first *formally verified* C compiler, in the sense that much of the compiler is written and proven correct in the Coq proof assistant.[3] Whether to trust Coq more than a carefully-developed C program is a personal decision (see Section 1.4), but the CompCert compiler is arguably more optimizing, extensible, and maintainable than previous verified compilers [25].

[1] In practice, CompCert seems reasonably time and space efficient [9, 25]

[2] http://www.windriver.com/

[3] The appendix in Section 7.1 gives a very short introduction to Coq.

1.1 Source Languages

CompCert supports most of the **ANSI C** and **ISO-C-99** standards. **ANSI C** refers to the American standard published in 1990 [10]. **ISO-C-99** refers to the international standard, published in 1999 [26]. (Suffice to say, these are widely used programming languages.) We will make reference to **ISO-C-99** as C99 throughout.

1.1.1 Details and Caveats

CompCert does not accept the entire C99 standard. These are the most interesting exceptions and design details, as documented in Chapter 5 of the CompCert reference manual [20].

- CompCert follows the *hosted environment* model, using the standard library of the target system. C99 defines a hosted implementation as one that accepts any strictly-conforming program [26]. In contrast, a *free-standing* implementation can reject programs that use standard headers other than `float.h, iso646.h, limits.h, stdarg.h, stdbool.h, stddef.h,` and `stdint.h`.
- Programs must be sequential. No `pthreads`!
- Preprocessing is done externally, by the system C compiler.
- Multibyte characters are not allowed in source code; for example λ.
- The `long double` type[4] and functions returning struct or union types are not allowed by default.
- Complex types (like `double _Complex`) and variable-length arrays are not supported.
- The `restrict` keyword is ignored. Normally, this keyword marks a variable as the exclusive accessor for its underlying object.
- The `switch` statement is the same from Java and Misra-C [7, 2]. In particular, Duff's Device is not allowed (see Appendix 7.1 for a definition).
- The `asm` statement (for inline assembly) may be enabled with a flag, and may be used to invalidate correctness guarantees.
- The `_Alignof` and `_Alignas` operators from **ISO-C-2011** are supported.
- CompCert assumes that `setjmp` and `longjmp` respect control flow. All local variables that are live across an invocation of `setjmp` must be declared `volatile` to avoid mis-optimization [20].
- The bit-level representation of integers and floats is exposed, but pointers are opaque 32-bit words.
- Integers follow the ILP32LL64 [**?**] model and floats follow the IEEE-754-2008 [8] standard. To paraphrase, arithmetic must be done with infinite precision and finally rounded to a machine-dependent precision when stored to a variable.
- The compiler will optimize floating point operations assuming that all floats are rounded to the nearest representable number, with ties broken by rounding to an even mantissa. Use a flag to disable this optimization if the rounding technique will change at runtime.
- The compiler will *not* implicitly reorder floating-point computations. Operators like fused multiply-add are available as primitives and must be called explicitly.

Programs written in C are preprocessed, parsed, type-checked, and elaborated into the Clight language, for which CompCert has formal semantics.

[4] CompCert can accept `long double` as a synonym for `double`. This compiles with C99, but may conflict with other binaries on some platforms.

1.2 Target Languages

CompCert provides 3 code generators [20]:

- PowerPC 32 bits (all models)
- ARM v6 and above with VFP coprocessor (for floating-point arithmetic)
- Intel/AMD x86 in 32-bit mode, with SSE2 extension (for integer-vector operations)

PowerPC is supported on all 32-bit Linux distributions. ARM is supported on Debian & Ubuntu Linux (the `armel` and `armhf` architectures. x86 works on all 32-bit and all 64-bit-emulating-32-bit Linux distributions, along with MacOS ≥ 10.6 and Windows via Cygwin. The specific application binary interfaces and related limitations are documented in the reference manual [20].

1.3 Compiler Structure

CompCert is implemented in 25 passes [20]. The first 3 are pre-processing: these convert C source code to CompCert C syntax. The next 19 are the compiler back-end, including 8 optimizing passes. The final 3 convert ASM syntax trees to raw assembly code, assemble an object file, and link to create an executable.

Of these passes, the parsing pass and 15 back-end passes are implemented and verified in Coq [20]. These constitute 90% of the compiler. File inclusion, macro expansion, conditional compilation, and other pre-processing steps are handled by an external C preprocessor. Assembling and linking are also handled externally; however, PowerPC users can use the `checklink` tool to assert basic properties about the linked code (i.e., linking did not remove any code blocks).

1.3.1 Optimizing Passes

These are the main optimizations, listed in the order that they appear as compiler passes. See Section 2 for details.

1. Instruction selection
2. Tail call elimination
3. Function inlininng
4. Constant propogation
5. Common subexpression elimination
6. Dead code elimination
7. Branch tunneling
8. Register allocation

The compiler does not perform any loop optimizations [20]. No hoisting, no reordering, no unrolling.

1.4 Guarantees

The compiler correctness theorem states, informally,[5] that compiled code *refines* the *behavior* of source code [20]. By *refines*, we mean that all compiled behaviors are observable source behaviors with the exception that compiled code may eliminate dead code that caused the source code to diverge. By *behavior*, we mean the trace of all I/O and volatile (read: mutable) operations plus the program's exit code, if any.

The behavior of source and compiled code is specified in terms of the languages' formal semantics. As C99 does not have a formal semantics, much less one implemented in Coq, we cannot even state a whole-compiler correctness theorem. Current research aims to fix this issue [13, 14].

[5] Section 3 introduces necessary terminology and Section 4 formally states the correctness theorem.

1.4.1 Why should we trust Coq?

CompCert's correctness theorem is stated and proved in Coq. If Coq is incorrect, then the theorem does not hold. However:

- The Coq developers believe in Coq
- The CompCert developers believe in Coq
- The Airbus team believes in CompCert
- John Regehr hasn't been able to break CompCert

Speaking of John Regehr, his group famously wrote in their PLDI'11 paper [27]:

> "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."

More recent bug-finding tools [15] have been unable to break the verified portions of CompCert as well. Even if we do not fully trust Coq, we should at least prefer it over hand-validated C.

1.5 Installing CompCert

CompCert is not free software, but a non-commercial version is on Github for "educational, research or evaluation purposes only". After cloning the repository:

1. `./configure ia32-linux`
   (MacOS and Cygwin users: change to `ia32-macosx` or `ia32-cygwin`)
2. `make`
3. `make install`

You can then compile a program using the `ccomp` executable. The interface to `ccomp` is nearly the same as `gcc`, only restricted to the options that CompCert implements.

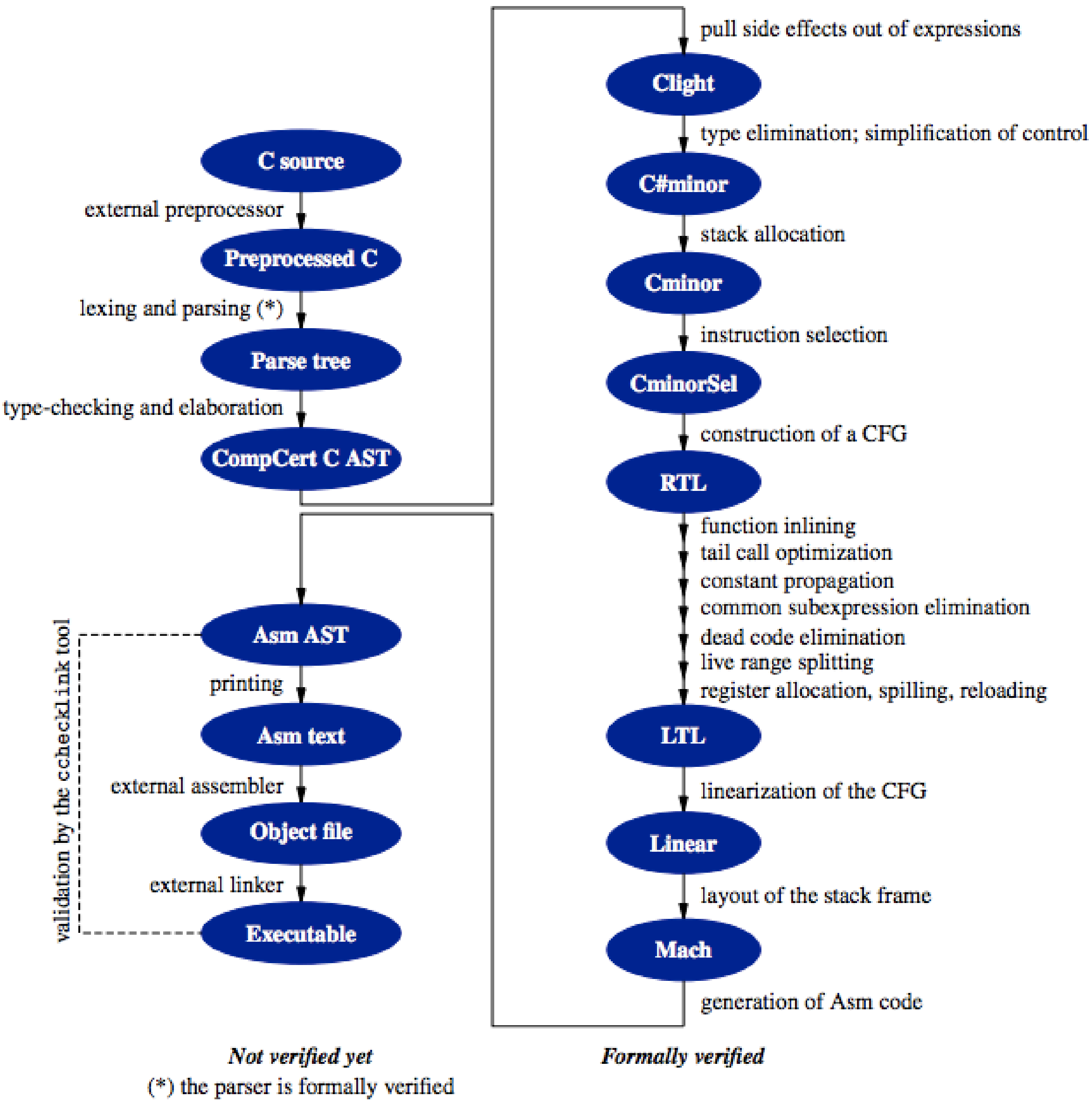

Figure 1: General structure of the CompCert C compiler [20]

2 From C to Asm

The CompCert compiler is divided across 25 passes. Figure 1.5 shows 21 of these; the missing passes are additional optimizations.

In the beginning there is C99; the end is PowerPC, ARM, or x86 32-bit assembly. Here we follow the path of C programs down to assembly.[6]

Following the diagram, this section is divided into 3 parts: front-end, back-end, and assembly. Our final reference for each pass is the annotated Coq development [19]. Inline hyperlinks point to specific lines in the Coq source; these may go stale, but are relevant as of November 19, 2015.

2.1 C Front-End

"CompCert C" programs are normal C programs. Case in point, we start with classic fibonacci:

```
#include <stdlib.h>
#include <stdio.h>

#define NDEFAULT 36

int fib(int n) {
  if (n < 2)  return 1;
  else  return fib(n-1) + fib(n-2);
}

int main(int argc, char ** argv) {
  int n, r;
  if (argc >= 2) n = atoi(argv[1]); else n = NDEFAULT;
  r = fib(n);
  printf("fib(%d) = %d\n", n, r);
  return 0;
}
```

The libraries `stdlib.h` and `stdio.h` are the system-standard C libraries, not specialized CompCert versions. The `#define` directive is expanded by a preprocessor. The `main` function accepts a variable number of arguments on the command line. Calling `printf` prints a string to the console. Even the command to compile this program is (nearly) indistinguishable from the `gcc` equivalent.

```
ccomp -o fib fibonacci.c        vs.        gcc -o fib fibonacci.c
```

The resulting executables are also similar, though the CompCert version runs slightly faster[7] due to having at least some optimizations. Changing to `gcc -O2` leaves CompCert in the dust; it is almost 50% faster.

Point being, CompCert C really is just a subset of familiar C and many programs As stated in Section 1, the subset does not include threaded programs or `setjump` or `long double` types, but it is a large subset nonetheless.

2.1.1 Preprocessing

Compilation begins with preprocessing. CompCert invokes a system preprocessor (determined in configuration and registered when building CompCert) to interpret macros & directives. There is no formal guarantee that the preprocessor's output matches the source code.

[6] `ia-32`, specifically, but the story for PowerPC and ARM is similar and, according to the authors, more efficient. "The IA32 architecture, with its paucity of registers and inefficient calling conventions, is not a good fit for the CompCert compilation model." [20]

[7] 5% faster, averaging over 10 runs of `./fib 52`.

2.1.2 Lexing and Parsing

After preprocessing, the entire C source file is loaded into memory and sent to a parsing automaton. The lexer and parser automata are generated by the Menhir parser generator [23] from a C grammar.

Lexing is apparently unverified, but the parser is guaranteed faithful to the grammar from which it was generated. This means that the parser accepts the same language as the specification grammar. Assuming the grammar correctly recognizes C99, we have a verfied C99 parser.

2.1.3 Pragmas

The parsed C AST is sent through a "C2C" tranformation to typecheck the AST and expand CompCert pragmas. This pass is implemented in OCaml and does not have a formal guarantee.

The types are simple C types.

```
Inductive type : Type :=
  | Tvoid: type
  | Tint: intsize -> signedness -> attr -> type
  | Tlong: signedness -> attr -> type
  | Tfloat: floatsize -> attr -> type
  | Tpointer: type -> attr -> type
  | Tarray: type -> Z -> attr -> type
  | Tfunction: typelist -> type -> calling_convention -> type
  | Tstruct: ident -> attr -> type
  | Tunion: ident -> attr -> type
```

The pragmas guide linker execution [20].

- `reserve_register` : prevent the compiler from using a specific register in compiled code.
- `section` : defines a linker section.
- `use_section` : explictly place definitions inside a declared section.

After this phase, we have a *CompCert C* AST and are ready to begin verified compilation.

2.2 Back-End

Every pass in this subsection has a Coq proof of correctness. All passes and proofs are online and annotated [19].

Note: Clight is the front-end described in the CompCert journal and conference publications [3, 16]. Officially, the back-end starts with `Cminor`, but we use "back-end" here to group the formally verified core passes of the compiler.

2.2.1 CompCert C to Clight

The first verified pass removes side effects out from expressions and into assignment statements. For example:

```
int x = 0;
return  (10 / x) + (x = 1);
```

becomes ...

```
int x, y;
x = 0;
y = 1;
x = y;
return (10 / x) + y;
```

The translation happens inside a monad for catching errors and generating fresh names (like `y`). If the pass is successful, CompCert asserts that the result is a valid Clight AST and moreover the source and result are related by a *forward simulation*. (Section 3 will explain the precise meaning of this and other proofs.)

```
Theorem transl_program_correct:
  forward_simulation (Cstrategy.semantics prog) (Clight.semantics1 tprog).
```

2.2.2 Simplify Locals (Clight to Clight)

The next pass moves *local scalar variables* whose address is not taken into temporary variables. The overall goal of this pass is to ensure that functions are only called on temporaries. Thus if a variable's address is ever taken *and* the variable is used as a function parameter, the function call is rewritten to reference a new temporary variable. In the end, all function parameters live on the stack when the function call happens.

This pass's correctness theorem relates two version of the Clight semantics; in `semantics2`, function parameters may only be temporaries.

```
Theorem transf_program_correct:
  forward_simulation (semantics1 prog) (semantics2 tprog).
```

2.2.3 Clight to C#minor

Next, we translate to the C#minor intermediate language. The pass resolves all type-dependent behaviors, including overloaded function applications and address computations, and converts loop and `switch` statements to slightly simpler representations. The pass guarantees that (second version of) the Clight semantics are in forward simulation with the C#minor semantics.

```
Theorem transl_program_correct:
  forward_simulation (Clight.semantics2 prog) (Csharpminor.semantics tprog).
```

2.2.4 C#minor to Cminor

C#minor functions may take the address of variables, but Cminor functions have exactly 1 addressable variable (the stack). This pass moves variables to the stack and converts address lookups to stack lookups.

```
Theorem transl_program_correct:
  forward_simulation (Csharpminor.semantics prog) (Cminor.semantics tprog).
```

Cminor is the last machine-independent intermediate language. This makes Cminor a reasonable target language for other high-level compilers that want to re-use CompCert's optimizations and assembly. Recent work on separation logic for Clight [1] and a static analyzer for C#minor [12] have given even more reasons to target Cminor.

2.2.5 Instruction Selection (Cminor to CminorSel)

The first machine-dependent pass recognizes sequences of operations that the target machine can perform in one instruction. On `ia-32`, the sum `n1 + n2` will translate to an immediate-add instruction if one of the operands is a constant. This pass also converts loads and stores to the target architecture's addressing modes.

The CminorSel semantics are tailored to a particular target language, but the overall theorem is similar to earlier passes.

```
Theorem transf_program_correct:
  forall prog tprog,
  sel_program prog = OK tprog ->
  forward_simulation (Cminor.semantics prog) (CminorSel.semantics tprog).
```

The pass will fail in the translation step `sel_program` if the target architecture is missing required operations (in particular, 64-bit arithmetic).

2.2.6 CFG generation (CminorSel to RTL)

After instruction selection, CompCert builds a control-flow graph of the program. The pass includes a heuristic based on the size of programs for choosing which branch of an `if`-statement should be the fall-through case.

```
| Sifthenelse c strue sfalse =>
    if more_likely c strue sfalse then
      do nfalse <- transl_stmt map sfalse nd nexits ngoto nret rret;
      do ntrue  <- transl_stmt map strue  nd nexits ngoto nret rret;
         transl_condexpr map c ntrue nfalse
    else
      do ntrue  <- transl_stmt map strue  nd nexits ngoto nret rret;
      do nfalse <- transl_stmt map sfalse nd nexits ngoto nret rret;
         transl_condexpr map c ntrue nfalse
```

Again, the proof of correctness is by forward simulation.

```
Theorem transf_program_correct:
  forward_simulation (CminorSel.semantics prog) (RTL.semantics tprog).
```

The RTL[8] language is very simple: just a graph of instructions. From here we start the heavier optimizing passes of CompCert.

2.2.7 Tail Call Recognition (RTL to RTL)

Converts some function calls into *tail calls*. A tail call does not require allocating a new stack frame, saving time and space. This is possible when the result of the function call is ignored or used immediately, provided the current function has no addressable variables.

CompCert also detects the more complicated tail call patterns, and optimizes the below to a tail call.

```
x = f(...);
nop;
y = x;                becomes ...        return f(...);
z = y;
return z;
```

This optimization complicates the proof of correctness, since the source and target versions of RTL might not perform the same transitions. Instead, if a step in the source was optimized away by the target, a well-founded measure shows that the target code performs fewer function calls.

[8] RTL stands for "register transfer language". The name is historical. GCC has an RTL intermediate language and textbooks often use an RTL [21].

```
Definition niter
Definition measure (st: state) : nat :=
  match st with
  | State s f sp pc rs m => (List.length s * (niter + 2) + return_measure f.(
     fn_code) pc + 1)%nat
  | Callstate s f args m => 0%nat
  | Returnstate s v m => (List.length s * (niter + 2))%nat
  end.
```

Additionally, the relation on source and target states allows the target to use more stack frame slots, provided the existing data is unchanged.

The overall proof is another forward simulation.

```
Theorem transf_program_correct:
  forward_simulation (RTL.semantics prog) (RTL.semantics tprog).
```

2.2.8 Function Inlining (RTL to RTL)

Function inlining rewrites functions declared as `inline` at their call site. There is no smart heuristic or measure of fuel to guide inlining. If the `inline` keyword is present, the function body is placed in the code.

There are two caveats:

- Inlining increases the size of a function's stack block. If the new block is more than the largest representable integer, compilation fails.
- Recursive functions are inlined exactly once—recursive calls are never inlined.

Inlining decreases the number of states in the CFG, so another well-founded measure describes how the target state is "less than" the source in case there is no longer a target step matching the source step. The overall theorem is the same as the last pass: RTL semantics are preserved.

```
Theorem transf_program_correct:
  forward_simulation (RTL.semantics prog) (RTL.semantics tprog).
```

2.2.9 Renumbering the AST (RTL to RTL)

Upcoming optimizations rely on Kildall's algorithm for static analysis. These analyses all proceed by propogating known information from a node in the CFG to its successors; the process repeats until reacing a fixed point.

If CFG nodes are numbered in post-order with respect to their edges, the static analysis phases run a little faster and more efficiently. This renumbering pass ensures a post-order numbering; it happens after inlining and tail calls because those passes change the structure of the CFG.

Correctness is by forward simulation.

```
Theorem transf_program_correct:
  forward_simulation (RTL.semantics prog) (RTL.semantics tprog).
```

2.2.10 Constant propogation (RTL to RTL)

Pushes known constants through expressions and assignments. For example:

```
int x = 4                                              int x = 4
  , y = 7 + x + x                                        , y = 15
  , z = (2 * x) + y + y + 1;      becomes ...            , z = 39;
return z + x - y;                                      return 28;
```

Constant propogation will also replace operators like + with cheaper versions if some, but not all, arguments are statically known.

Proof is by forward simulation. In this case, the well-founded measure is step indexing.

```
Theorem transf_program_correct:
  forward_simulation (RTL.semantics prog) (RTL.semantics tprog).
```

2.2.11 Common subexpression elimination (RTL to RTL)

Avoids redundant computations. If two variables are assigned to the same expression, this pass replaces the later use with a variable reference—like memoization.

```
int r1 = a + b;                  int r1 = a + b;
int r2 = a + b     becomes ...   int r2 = r1;
```

CompCert's CSE is not very powerful. Small variations in the code will trick it:

```
int r1 = a + b;                  int r1 = a + b;
int r2 = b + a     becomes ...   int r2 = b + a;
```

and function calls are never optimized.

Still, the optimizations that are implemented are also proven to preserve the RTL semantics. This pass does not change the number of program steps; one transition in the source is always matched by a transition in the target.

```
Theorem transf_program_correct:
  forward_simulation (RTL.semantics prog) (RTL.semantics tprog).
```

2.2.12 Dead Code Elimination (RTL to RTL)

Changes unreachable, or otherwise useless, instructions into no-ops. This pass can change termination behavior:

```
int y = 4 / 0;                  noop;
return 4;         becomes ...   return 4;
```

CompCert reasons that the source program in this case is *unsound* and allows dead code elimination to proceed. This reasoning is used within the simulation proof. Otherwise, the proof statement is straightforward. Each source step is matched exactly with a target step; however the matching target step is more often a no-op.

```
Theorem transf_program_correct:
  forward_simulation (RTL.semantics prog) (RTL.semantics tprog).
```

2.2.13 Unused Globals (RTL to RTL)

Removes unused global variables from the environment. Source and target steps remain synchronized; the proof only guarantees that steps did not depend on one of the removed globals.

```
forward_simulation (semantics p) (semantics tp).
```

2.2.14 Register allocation (RTL to LTL)

Register allocation determines which program variables can fit in machine registers, and which must be stored in memory. For each function in the program, CompCert uses George & Appel's *iterated register coalescing* algorithm to assign variables to registers [6].

This algorithm works by:

- Splitting the CFG into *live ranges* for variables.
- Constructing a graph of dependencies between live ranges.
- Using a heuristic solver to color the graph. If coloring succeeds, finish.
- If coloring fails, insert store & load instructions to relax the graph and repeat from step 2.

Rather than implement the algorithm in Coq and directly prove its correctness, the CompCert authors use an OCaml implementation. At runtime, the (extracted) Coq development gives an RTL function AST to the OCaml procedure. The OCaml code attempts to find a register allocation and either returns a failure code or a solution in the form of an LTL function AST. Finally, the Coq development validates the result AST against the original function [24]. If the result is correct, this pass succeeds.

The overall proof for this pass is another forward simulation. In this case, one source step is matched by one or more target steps.

```
Theorem transf_program_correct:
  forward_simulation (RTL.semantics prog) (LTL.semantics tprog).
```

The result of this pass is an LTL program, meaning the program uses machine registers and stack slots rather than pseudo-registers. Control flow is partially linearized, as graph nodes are now basic blocks instead of instructions.

2.2.15 Branch Tunneling (LTL to LTL)

Tunneling converts a sequence of jumps with no intermediate instructions into one long jump to the target. In psuedocode:

```
L1; nop; L2;                  L1; nop; L4;
L2; nop; L3;    becomes ...   L2; nop; L4;
L3; nop; L4;                  L3; nop; L4;
```

The proof is a forward simulation within LTL. When the target cannot match a source transition, the target has strictly fewer jumps than the source.

```
Theorem transf_program_correct:
  forward_simulation (LTL.semantics prog) (LTL.semantics tprog).
```

2.2.16 Linearization (LTL to Linear)

Linearization converts our LTL control-flow graph into a flat sequence of instructions. Jump instructions replace the graph's edges, except where jumping gives the same result as falling through to the next instruction.

Minimizing the number of explicit jumps is an optimization challenge: pick an ordering on code blocks that maximizes the number of fall-throughs. Once again, an OCaml procedure applies a heuristic to pick a block ordering and a Coq function folds over the blocks to create a sequence of instructions.

The simulation proof for this pass uses a measure on the number of basic blocks in the program. If the target cannot match a source transition, then the source transition must have just exited a block.

```
Theorem transf_program_correct:
  forward_simulation (LTL.semantics prog) (Linear.semantics tprog).
```

2.2.17 Label Pruning (Linear to Linear)

Linearization always produces a label for each LTL node. This pass removes unused labels. (The extra labels could complicate future optimizations, but CompCert has none of those yet.)

Proof is by forward simulation. Target code must match each source step or show that the source transition moved forward along the current block.

```
Theorem transf_program_correct:
  forward_simulation (Linear.semantics prog) (Linear.semantics tprog).
```

2.2.18 Activation Record (Linear to Mach)

This pass conforms all call and return points in the linearized code to the target machine's calling conventions.

Proof is by forward simulation, where each source transition is matched by one or more target transitions.

```
Theorem transf_program_correct:
  forward_simulation (Linear.semantics prog) (Mach.semantics
     return_address_offset tprog).
```

2.2.19 Assembly Generation (Mach to Assembly)

The final verified pass of the compiler produces an assembly AST. There is one Coq implementation for each of the three possible backends: PowerPC, ARM, and `ia-32`.

The proof is by forward simulation; the correspondence between source and target is nearly one-to-one, but Mach may take an extra no-op step from a return state to the next state.

```
Theorem transf_program_correct:
  forward_simulation (Mach.semantics return_address_offset prog) (Asm.
     semantics tprog).
```

2.3 Assembly-End

The final passes of the compiler turn a Coq model of assembly code into executable code. These passes are unverified, though on PowerPC the cchecklink tool will scan for basic translation errors like mangling a data segment or dropping function definition.

2.3.1 Printing (Assembly to Text)

An OCaml function prints the assembly AST, converting all the Coq constructors into register and instruction names.

2.3.2 Assembling

An external assembler collates the text into an object file. On my machine, this is `gcc`.

2.3.3 Linking

An external linker combines this object file with any others in the compilation pass, or with separately compiled files. Again, my machine uses `gcc`.

That's all there is to it!

3 Speaking of Compiler Correctness

3.1 Abstract Correctness

CompCert is a compiler. As a compiler user, CompCert translates my C99 programs to executable files.

$$\text{C99} \xrightarrow{\hspace{4em}\texttt{ccomp}\hspace{4em}} \text{Exe}$$

This translation is correct if the executable performs the same sequence of operations that the source program *should* perform, if I was to interpret it line-by-line. For instance, the executable for the program on the left should print to the console exactly once and exit cleanly. The executable for the program on the right should print to the console until I forcibly kill the program.

```
int main(void) {
  printf("Hello\n");
  return 0;
}
```

```
int main(void) {
  while (1) printf("Hello\n");
  return 0;
}
```

More precisely, a correct compiler will translate the semantics of the source program into equivalent semantics in the target program. There are generally a few ways to specify this notion of *semantics preservation*. CompCert uses simulations.

3.2 Notions of Simulation

Let $\mathbb{C}$ be a compiler from source language $\mathbf{S}$ to target language $\mathbf{T}$. Suppose $\mathbf{s}$ is a program written in language $\mathbf{S}$ and $\mathbf{t}$ is the result of compiling $\mathbf{s}$. In other words, $\mathbf{t} = \mathbb{C}(\mathbf{s})$.

Suppose the semantics for S imply that $\mathbf{s}$ should begin at state $\mathbf{s}_0$, run for n steps, and halt at the final state $\mathbf{s}_n$. Also suppose that $\mathbf{t}$ begins at state $\mathbf{t}_0$, runs for m steps, and finishes at state $\mathbf{t}_m$. Some of the transitions on either side may cause an observable effect, like printing to the screen or closing a file, so we label each edge with a tag e describing the effect (if any). Our general picture is now:

$$\begin{tikzcd}
\mathbf{s}_0 \arrow[d, "e_{s1}"'] & & \mathbf{t}_0 \arrow[d, "e_{t1}"] \\
\mathbf{s}_1 \arrow[d, "e_{s2}"'] & & \mathbf{t}_1 \arrow[d, "e_{t2}"] \\
\vdots \arrow[d, "e_{sn}"'] & & \vdots \arrow[d, "e_{tm}"] \\
\mathbf{s}_n & & \mathbf{t}_m
\end{tikzcd}$$

Our goal is to prove by simulation that $\mathbf{s}$ and $\mathbf{t}$ are equal programs based on their transition structure.

3.2.1 Bisimulation

A straightforward technique is to demonstrate a *bisimulation* relation between the source and target. This amounts to giving a relation R such that:

- $(\mathbf{s}_0, \mathbf{t}_0) \in R$
- For all states $\mathbf{s}_i$, $\mathbf{s}_i \xrightarrow{e_{s(i+1)}} \mathbf{s}_{i+1}$ implies that for all target states such that $(\mathbf{s}_i, \mathbf{t}_j) \in R$, there exists a target state $\mathbf{t}_{j+1}$ such that $\mathbf{t}_j \xrightarrow{e_{t(j+1)}} \mathbf{t}_{j+1}$ and $(\mathbf{s}_{i+1}, \mathbf{s}_{j+1}) \in R$. Also, $e_{s(i+1)}$ and $e_{t(j+1)}$ are the same effect.

- The reverse holds for target transitions: For all states $\mathbf{t}_j$, $\mathbf{t}_j \xrightarrow{e_{t(j+1)}} \mathbf{t}_{j+1}$ implies that for all source states such that $(\mathrm{s}_i, \mathbf{t}_j) \in R$, there exists a source state s_{i+1} such that $\mathrm{s}_i \xrightarrow{e_{s(i+1)}} \mathrm{s}_{i+1}$ and $(\mathrm{s}_{j+1}, \mathbf{t}_{j+1}) \in R$ Also, $e_{s(i+1)}$ and $e_{t(j+1)}$ are the same effect.

In terms of the picture above, the proof would require $n = m$ and for each $i \in \{0..n\}$, $(\mathrm{s}_i, \mathbf{t}_i) \in R$.

3.2.2 Backward Simulation

Bisimulation is a very strong notion of equivalence, but it definitely shows that s and $\mathbf{t}$ are equal, in the sense that every action of the source is matched by an action in the target, and vice-versa. It is so strong that it disallows basic compiler optimizations: if the source program contains no-op instructions, the compiled version must take the same number of do-nothing transitions. A *backward simulation* $R_{\leftarrow}$ allows these optimizations. The requirements are:

- $(\mathrm{s}_0, \mathbf{t}_0) \in R_{\leftarrow}$
- All target transitions are matched by *one or more* source transitions. For all states $\mathbf{t}_j$, $\mathbf{t}_j \xrightarrow{e_{t(j+1)}} \mathbf{t}_{j+1}$ implies that for all source states such that $(\mathrm{s}_i, \mathbf{t}_j) \in R_{\leftarrow}$, there exists a source state s_k such that $\mathrm{s}_i \xrightarrow{e_{sk}} {}^{+}\mathrm{s}_k$ and $(\mathrm{s}_k, \mathbf{t}_{j+1}) \in R_{\leftarrow}$. Also, e_{sk} and $e_{t(j+1)}$ are the same effect.

The notation $\longrightarrow^{+}$ denotes one or more transitions.[9] This way, our compiler is free to make $\mathbf{t}$ a more efficient version of s.

3.2.3 Forward Simulation

Another useful notion of simulation is to remove the third bisimulation requirement, but keep the second. As with backwards simulations, it is useful for *forward simulations* to accept multiple target steps in response to one source step.

- $(\mathrm{s}_0, \mathbf{t}_0) \in R_{\rightarrow}$
- For all states s_i, $\mathrm{s}_i \xrightarrow{e_{s(i+1)}} \mathrm{s}_{i+1}$ implies that for all target states such that $(\mathrm{s}_i, \mathbf{t}_j) \in R_{\rightarrow}$, there exists a target state $\mathbf{t}_k$ such that $\mathbf{t}_j \xrightarrow{e_{tk}} \mathbf{t}_k$. Also, $e_{s(i+1)}$ and e_{tk} are the same effect.

3.2.4 Measured Forward Simulation

CompCert proves many of its optimizing passes using forward simulations augmented by a well-founded measure. The general shape of these proofs is to use a measure $<$ on source states to prove a relation $R_{\rightarrow}$ such that:

- Given a source program s and compiled version $\mathbf{t}$, we have $(\mathrm{s}, \mathbf{t}) \in R_{\rightarrow}$
- Whenever the source program takes a step to s′, either:
  - The target program takes a step, and the results of each step are related by $R_{\rightarrow}$
  - The target program does *not* step, but $(\mathrm{s}', \mathbf{t}) \in R_{\rightarrow}$ and $\mathrm{s}' < \mathrm{s}$.

The last scenario is a *stutter* in the source. The measure $<$ ensures that we do not stutter forever, but rather make progress at some point. As an example, $<$ might count the number of no-ops in a program and $\mathbf{t}$ might be a version of s with all no-ops removed. It should be easy to show that all transitions of source and target match, except in the case that the source program performs a no-op.

[9] Allowing a sequence of transitions means we need to compose effects. For simplicity, assume that the only way to compose effects is to delete a null effect from the left or right end of a sequence.

3.2.5 Composition

CompCert's proof of correctness relies on two properties of simulations. We name and sketch proofs of these simulations, assuming for simplicity that there are no effects and no skipping transitions. Full proofs are in the CompCert repository.

Forward Simulations Compose [proof] Suppose we have two forward simulations: one from $\mathbf{S}$ to $\mathbf{T}$ and the other from $\mathbf{T}$ to $\mathbf{T}'$. Call these R_1 and R_2, respectively. We claim that $R_3 = \{\, (\mathbf{s}_i, \mathbf{t}'_k) \mid (\mathbf{s}_i, \mathbf{t}_j) \in R_1 \wedge (\mathbf{t}_j, \mathbf{t}'_k) \in R_2 \,\}$ is a forward simulation from $\mathbf{S}$ to $\mathbf{T}'$.

First, all initial states of $\mathbf{S}$ and $\mathbf{T}'$ are related by construction: if $\mathbf{s}_0$ is an initial state, then it must be related by R_1 to some $\mathbf{t}_0$ and likewise R_2 must relate $\mathbf{t}_0$ to some initial state $\mathbf{t}'_0$. Therefore R_3 relates $\mathbf{s}_0$ to $\mathbf{t}'_0$. Second, if $\mathbf{s}_i$ steps to $\mathbf{s}_{i'}$ and $(\mathbf{s}_i, \mathbf{t}_j) \in R_1$ then:

- There exists a $\mathbf{t}_{j'}$ such that $\mathbf{t}_j \to \mathbf{t}_{j'}$ and $(\mathbf{s}_{i'}, \mathbf{t}_{j'}) \in R_1$
- If there is an $\mathbf{t}'_k$ such that $(\mathbf{t}_j, \mathbf{t}'_k) \in R_2$ then there exists a $\mathbf{t}'_{k'}$ such that $\mathbf{t}'_k \to \mathbf{t}'_{k'}$ and $(\mathbf{t}_{j'}, \mathbf{t}'_{k'}) \in R_2$).
- By construction, $(\mathbf{s}_i, \mathbf{t}'_k) \in R_3$
- The state $\mathbf{t}'_{k'}$ matches the transition from $\mathbf{s}_i$ to $\mathbf{s}_{i'}$.

Forward Simulations imply Backward Simulations [proof] If we have a deterministic target language, then a forward simulation from source to target implies a backward simulation from target to source. Suppose $(\mathbf{s}_i, \mathbf{t}_j)$ are related source and target states and furthermore $\mathbf{t}_j \to \mathbf{t}_{j'}$. We know that $\mathbf{t}_{j'}$ is unique because the semantics is deterministic. So it must be true that all transitions out of $\mathbf{s}_i$ match the transition from $\mathbf{t}_j$—otherwise, we would contradict our assumed forward simulation.[10]

3.3 Proving a Compiler

Let $\simeq$ be one of the above simulations. A verified compiler $\mathbb{C}$ guarantees that for all source programs $\mathbf{s}$, if $\mathbb{C}(\mathbf{s}) = \mathbf{t}$ then $\mathbf{s} \simeq \mathbf{t}$. Note that if the compiler fails to produce a result, we have no $\mathbf{t}$ and no theorem. Making sure that a compiler successfully compiles all valid source programs is an important engineering issue [17].

One way to verify a compiler is to reason directly about its source code. This is straightforward, but potentially difficult and less efficient [4] than an unverified compiler.

An alternative is to use an unverified compiler along with a *translation validator* $\mathbb{V}$, and abort compilation if the validator rejects the compiler's results. The guarantee would then be: for all source programs $\mathbf{s}$, if $\mathbb{C}(\mathbf{s}) = \mathbf{t}$ and $\mathbb{V}(\mathbf{t}) = \mathsf{true}$ then $\mathbf{s} \simeq \mathbf{t}$. This approach shifts the proof effort to the validator. If one can show that the validator correctly recongizes valid and invalid results, we have a verified compiler.

As we shall see, CompCert uses both techniques.

[10] I didn't fully understand the Coq proof, but as far as I can tell it is classical. Being classical is not such a big deal because there are finitely many transitions out of each state.

4 CompCert's Correctness

Our picture of CompCert from Section 3 was:

$$\begin{tikzcd}\text{C99} \arrow[rr, "\texttt{ccomp}"] & & \text{Exe}\end{tikzcd}$$

but CompCert does not guarantee that every executable it generates simulates the original C99 program. For one, the C99 standard does not include a formal semantics. It leaves some behaviors undefined, like the order that function arguments are evaluated. Also the executable does not have formal semantics modeled in Coq, and all CompCert's guarantees are Coq theorems. The true picture is:

$$\begin{tikzcd}\text{C99} \arrow[r, dashed, "\texttt{gcc}"] & \text{Csem} \arrow[rr, "\texttt{ccomp}"] & & \text{Asm} \arrow[r, dashed, "\texttt{gcc}"] & \text{Exe}\end{tikzcd}$$

Where Csem is CompCert's semantics for C99 (after preprocessing) and Asm is CompCert's model of assembly language.[11] The dashed arrows are not verified, with the exception of the parsing stage (see Section 5.3, below). Moreover, the solid arrow from Csem to Asm is divided into many smaller arrows; one for each pass described in Section 2.

4.1 Caveats

Compiler correctness is built from small proofs of each pass, but before we can state the proper theorem we must address two pitfalls.

First, **non-determinism** can raise issues when it is present in either the target or source languages. Non-determinism in the *target* language makes forward simulation proofs less effective. Showing that all source behaviors are matched by a target behavior is good, but there could be additional target behaviors that shows up at runtime. (Non-determinism means there's a choice both at proof-time and run-time.) CompCert solves this by additionally proving that all target languages are deterministic.

Non-determinism in the *source* language conversely makes backward simulations less appealing, because the proof only needs to work for one of the possible source transitions. Unfortunately the non-determinism in C99 is unavoidable (at least until Airbus starts using the 2020 CompCert C standard). CompCert's position is to *choose one* of the source program's potential behaviors. This is documented the source code as CompCert's reduction strategy, as opposed to the valid semantics listed.

Second, **diverging** source programs present a difficulty. Should a verified compiler guarantee that all diverging source programs become diverging assembly programs? It is impossible to tell whether a source program will diverge or not, so preserving divergence would either require the compiler to be extremely faithful to the source program's semantics (read: optimize less) or force it to conservatively fail on more programs (because a verified compiler may reject as many programs as it likes). Neither solution is ideal; consider the dead-code optimization:

```
int y = 4 / 0;                   noop;
return 4;         becomes ...    return 4;
```

The source program diverges, but the target halts. Nonetheless, dead-code elimination is a pragmatic optimization.

CompCert takes a "garbage in, garbage out" stance. If the source program is unsafe or demonstrates undefined behavior, the correctness theorem does not hold. The only guarantees are for converging source programs.

4.2 CompCert's Correctness

Compiler correctness is stated as the composition of smaller theorems for each pass.

[11] Either PowerPC, ARM, or `ia-32`

- Each pass between C CompCert C (the determinisitic semantics) and Asm is proved using a forward simulation. Some of these simulations are measured, but others are in one-to-one correspondence.
- Forward simulations from RTL to Asm, from Cminor to Asm, and from Cstrategy to Asm are derived as the composition of pass simulations.
- Backward simulations for each of these 3 pairs of languages are derived from the forward simulations.
- The final theorem: Csem is in backwards simulation with the CompCert C strategy, and thereby with Asm.

We can paraphrase this result as "every observable Asm behavior is an observable target behavior".

The Coq theorem is:

```
Theorem transf_c_program_correct:
  forall p tp,
  transf_c_program p = OK tp ->
  backward_simulation (Csem.semantics p) (Asm.semantics tp).
```

4.3 Missing Theorems

CompCert does *not* prove some desirable properties:

- Semantics-preserving preprocessing
- Safe assembling, safe linking
- Whether compilation halts (the OCaml heuristics or unverified portions might loop forever)
- Whether the executable is free of segfaults or null pointer exceptions
- Time / space efficiency of the compiler
- Time / space efficiency of the generated code
- Anything about threaded programs

Some of these are failures of CompCert, others are failures of C. They are all opportunities for improvement.

5 Verified Toolchain

Compiler correctness from CompCert C to Assembly is a terrific achievement, but only the first milestone for the CompCert team. The original release did not support `goto` statements or union types, but these have been added over the years.

5.1 cchecklink

The `cchecklink` tool validates the results of assembling and linking on the PowerPC EABI and ELF/SVR4 platforms [20]. It works by comparing `.sdump` files generated by CompCert against the linker's executable. Every variable and function in the assembly code is matched against a code segment in the executable.

`cchecklink` does not claim to prove anything, but provides "an array of sanity checks for making sure the assembler/linker do not perform unsafe late-stage optimizations". This is weak assurance, but certainly better than having nothing at all (as is the case on ARM or `ia-32`).

The `checklink` sources are online in Robbert Krebber's mirror,[12] but not in the `AbsInt` repo. Besides the reference manual [20], `checklink` is undocumented.

5.2 Floating Point Arithmetic

CompCert includes a complete formalization of IEEE-754 floating-point arithmetic [8, 5]. From a users perspective, this means that floating point operations are essentially done with infinite precision and rounded when stored to variables—that is the IEEE specification. This guarantee is impressive because it holds for all three of CompCert's back-ends, regardless of whether the underlying machine is 32-bit or 64-bit.

In a lesser compiler, floating point numbers might be represented with double precision (using 64-bit numbers with 53-bit precision) or extended precision (80-bit numbers, 64-bit precision) *depending on the underlying machine*. At first glance, this should not be a problem because the C standard dictates that `float` variables use 32 bits and `double` variables use 64 bits, but there are two under-specified points. For one, the `long double` type uses a machine-dependent amount of precision. Second, it is not clear where to round variables in the middle of a compound expression. There are machine-specific and compiler-specific opportunities to get different results out of the same operation.

CompCert's solutions are to:

- Ban the `long double` type, allowing it only as a synonym for `double`.
- Never use extra precision bits.
- Round intermediate operations to the more-precise operand

Additionally, CompCert will never reorder instructions to reduce the number of multiply or add operations in a term. Instead it offers a library of "fast math" operations (like fused multiply-add) that the user may call explicitly.

Floating point arithmetic is nasty business. The mathematician Alston Householder once said that he "would never fly in an aircraft that had been designed with the help of floating point arithmetic".[13] Modern planes are now *flown* using floating-point arithmetic. Let's hope those programs were compiled with CompCert.

5.3 Verified Parsing

An important part of CompCert's front-end is converting C99 source code to an abstract syntax tree. This task is done by a parser created with the Menhir parser generator. Using a parser generator is important because parser automaton are often very large and difficult to read (CompCert's is over 60,000 lines) but the parser's spec should ideally be small enough to stand manually validation against a reference like the C99 standard.

[12] https://github.com/robbertkrebbers/compcert/tree/master/checklink

[13] http://homepages.gac.edu/~holte/talks/talk-flp.pdf

Menhir now guarantees that the parsers it generates accept the same language as the input grammar that generates them [11]. The proof is by translation validation: before declaring a successful parse, Menhir validates the result AST against the specification grammar. There is still a danger that the grammar does not recognize the C99 language, but the specification grammar is at least easier to manually check than the full parser. Moreover, the Menhir parser is reasonably efficient, only 3x slower than an unverified parser.[14] The net affect on compilation is within a 20% slowdown [11].

[14] http://gallium.inria.fr/blog/verifying-a-parser-for-a-c-compiler/

6 Commercial Users of Verified Software

The biggest recent news about CompCert was its partnership with AbsInt, a German company that sells static analysis tools. Xavier (and INRIA) is still in charge of CompCert research, but AbsInt now markets the compiler and provides technical support. AbsInt is also managing the CompCert github repository.

Even before the AbsInt partnership, Airbus was using the CompCert compiler to develop software that flies planes. Previously, Airbus had used the DIAB compiler managed by Wind River Systems. Airbus programs are still run through the DIAB preprocessor, assembler, and linker[15] but CompCert's verified optimizations made it a better choice than the DIAB compiler. As of February 2014, Airbus reported a 12% improvement in worst-case execution time as a result of using CompCert [25].

Providing safety guarantees for critical software was CompCert's goal from the beginning, but the partnerships raise an interesting question: should academic research strive to make a commercial partnership? Is there any other way to measure overall research impact than by commercial adoption?

Related point: CompCert is a giant engineering effort. Many people believed, back in 200X, that it might be possible to write a compiler in Coq. But that belief was not making embedded systems any safer. Xavier made it his *research* agenda to build a certified compiler. Was that a good use of time, or should "implementation details" be left to the industry folks?

To give two more examples of academia-to-industry projects:

6.1 Separation Logic

John Reynolds and Peter O'Hearn worked for years on developing a logic to express memory constraints. In 2009, Peter O'Hearn founded a startup, Monoidics, to build static analysis tools using the ideas from separation logic. Monoidics was bought by Facebook in 2013 and now the company's tools are validating the Android apps that Facebook publishes every month.

John Reynolds would have never implemented separation logic. Would we, or should we, think less of the research if it was not being used in practice?

6.2 GNU NYU Ada Interpreter (GNAT)

In 1992, NYU entered a contract with the US Air Force to write an Ada compiler. Ada was, and still is, a popular language for high-assurance software due to its stong type system and low-level capabilities. At any rate, the NYU team built both a compiler and an IDE for Ada. The project is now managed by a successful company, AdaCore.

Ada and C are less glamorous than Coq, OCaml, or Haskell. But they arguably have more *impact* on real software.

[15] DIAB has been around for over 25 years.

7 Learn More

7.1 OPLSS Tutorial

In 2011 and 2012, Xavier Leroy taught a short course on verified compilers at the Oregon Programming Languages Summer School.[16] The course presents a small compiler for the IMP programming language and proves a few optimizations.[17] To summarize:

- The source language has assignment statements, if statements, and while loops.
- The target language is a JVM-inspired stack language.
- Basic compiler correctness follows from two theorems.
  - If the source program terminates in state σ, then the compiled code will also terminate in σ.[18]
  - If the source program diverges, then so does the compiled code.
- Dead-code elimination (dce) is implemented by approximating the live variables in a program and removing all assignments to dead variables. (This is a source-to-source translation.) A correct dce pass ensures that optimized programs produce "the same" result state as un-optimized programs, where "the same" means two states hold the same values for all live variables.

  ```
  Theorem dce_correct_terminating:
    forall st c st’, c / st || st’ ->
    (* Program `c` started in state `st` evaulates to state `st’` *)
    forall L st1,                (* `L` is a set of live variables *)
    agree (live c L) st st1 ->   (* `live` computes a fixpoint of `L` *)
    exists st1’, dce c L / st1 || st1’ /\ agree L st’ st1’.
  ```

  For diverging programs, the correctness theorem includes a well-founded measure on commands and uses small-step semantics. Then an instruction either progresses to a new state or deducts from the measure.

- Register allocation is validated; the compiler checks that mapping on variables preserves semantics, taking special cases into account. (An external program should compute the allocation, but this is not part of the material.) To make register allocation and dead-code elimination compose, the statement of agree is generalized over a valid map between variables.
- Two static analyzers are implemented—one simple, the other with widening and inverse analysis of arithmetic and boolean operations. These analyzers are proven sound, and then used to implement constant propogation.

[16] https://www.cs.uoregon.edu/research/summerschool/

[17] As formalized in the Software Foundations textbook. [22]

[18] A state is just a map from identifiers to variables. Source and target code use the same representation for states.

Appendix A: Duff's Device

Here is Duff's Device, copied from Wikipedia.[19] CompCert will not compile this program.

```
int n = (count + 3) / 4;
switch (count % 4) {
case 0: do { *to = *from++;
case 3:      *to = *from++;
case 2:      *to = *from++;
case 1:      *to = *from++;
        } while (--n > 0);
}
```

The device is an optimized version of a typical do/while loop, performing up to 4 operations between branches.

```
do {
    *to = *from++;
} while(--count > 0);
```

The trick is interleaving the `do` statement with the cases in the `switch` statement. At the beginning, when the `switch` condition is evaluated, control jumps to the middle of the case statements depending on the value of `(count % 4)`. Control falls through one `case` to the next, so upon reaching the `while` condition the number of items left in `from` is divisible by 4. If the number is non-zero, we jump to the top of the `do` statement, perform 4 copies, and repeat the check.

This is sneaky and controversial code. One could argue that this is useful code, but MISRA-C and CompCert reject it.

[19] https://en.wikipedia.org/wiki/Duff%27s_device

Appendix B: Coq

The Coq proof assistant is built on the principle of *propositions as types*. A type specification in Coq is taken as a proposition, and a well-typed program is the corresponding proof.

This economy of concepts makes for a very powerful base language, but practical Coq programs are normally divided among conventional-looking programs and propositions about their correctness. The propositions are shown correct in a so-called *tactic language* designed to make writing proof terms easy.

Here is a sample Coq interaction that states and proves a few properties of the natural numbers. Coq includes many tools to make interactions like these easier, but hopefully the code gives a sense of the effort and lines-of-code vs. lines-of-proof breakdown required to develop CompCert. (Imagine specifying pointer and floating-point arithmetic!)

```
Inductive natural :=
  Zero : natural
| Add1 : natural -> natural.

Definition two  := Add1 (Add1 Zero).
Definition four := Add1 (Add1 (Add1 (Add1 Zero))).

Fixpoint plus (n1 : natural) (n2 : natural) : natural :=
  match n1 with
    | Zero => n2
    | Add1 n1’ => Add1 (plus n1’ n2)
  end.

Example two_and_two_make_four : plus two two = four.
Proof. auto. Qed.

Lemma plus_Zero_left : forall (n : natural), plus Zero n = n.
Proof. auto. Qed.

Lemma plus_Zero_right : forall (n : natural), plus n Zero = n.
Proof. induction n.
       auto.
       simpl. rewrite -> IHn. auto. Qed.

Lemma plus_associative :
  forall (n1 n2 n3 : natural), (plus (plus n1 n2) n3) = plus n1 (plus n2 n3).
Proof. intros n1 n2 n3. induction n1.
       auto.
       simpl. rewrite -> IHn1. auto. Qed.

Lemma plus_Add1_r :
  forall (n1 n2 : natural), plus n1 (Add1 n2) = Add1 (plus n1 n2).
Proof. intros n1 n2. induction n1.
       auto.
       simpl. rewrite -> IHn1. auto. Qed.

Lemma plus_commutative :
  forall (n1 n2 : natural), plus n1 n2 = plus n2 n1.
Proof. intros n1 n2. induction n1.
       rewrite -> plus_Zero_left. rewrite -> plus_Zero_right. auto.
       simpl. rewrite -> IHn1. rewrite -> plus_Add1_r. auto. Qed.
```

References

[1] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step cminor. In *TPHOLs*, 2007. http://www.ensiie.fr/~blazy/AppelBlazy07.pdf.

[2] Motor Industry Software Reliability Association. MISRA-C:2004 guidelines for the use of the C language in critical systems. 2004. http://caxapa.ru/thumbs/468328/misra-c-2004.pdf.

[3] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM*, 2006.

[4] Manuel Blum. On the size of machines. In *Information and Control*, volume 11, 1967. http://www.cs.cornell.edu/courses/cs6110/2015sp/docs/Blum-size-theorem.pdf.

[5] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. 2014. http://www.imm.dtu.dk/arith21/presentations/pres_64.pdf.

[6] Lal George and Andrew W. Appel. Iterated register coalescing. In *TOPLAS*, 1996.

[7] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java language specification. 2014. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[8] IEEE. Standard for floating-point arithmetic. 2008. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610935.

[9] INRIA. CompCert. http://compcert.inria.fr/index.html Accessed 2015-11-08.

[10] IT-015. AS 3955-1991. 1991. http://infostore.saiglobal.com/store/Details.aspx?ProductID=306911.

[11] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In *ESOP*, 2012. http://gallium.inria.fr/~xleroy/publi/validated-parser.pdf.

[12] Jaques-Henri Jourdan, Vincent Laporte, Sandrive Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL*, 2015. http://gallium.inria.fr/~xleroy/publi/verasco-popl2015.pdf.

[13] Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, 2015. http://robbertkrebbers.nl/research/thesis.pdf.

[14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C semantics: CompCert and the C standard. In *ITP*, 2014.

[15] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *POPL*, 2014. https://mehrdadafshari.com/emi/paper.pdf.

[16] Xavier Leroy. Formal verification of a realistic compiler. In *CACM*, 2009.

[17] Xavier Leroy. A formally verified compiler back-end. In *Journal of Automated Reasoning*, 2009.

[18] Xavier Leroy. Verified squared: Does critical software deserve verified tools? In *POPL*, 2011. Invited Talk.

[19] Xavier Leroy. The CompCert verified C compiler commented coq development. 2015. http://compcert.inria.fr/doc/index.html Last updated 2015-06-12.

[20] Xavier Leroy. The CompCert verified C compiler documentation and user's manual. 2015. http://compcert.inria.fr/man/manual.pdf Last updated 2015-06-11.

[21] Steven Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[22] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. with Loris D'Antoni and Andrew W. Appel and Arthur Azevedo de Amorim and Arthur Chargueraud and Anthony Cowley and Jeffrey Foster and Dmitri Garbuzov and Michael Hicks and Ranjit Jhala and Greg Morrisett and Jennifer Paykin and Mukund Raghothaman and Chung-chieh Shan and Leonid Spesivtsev and Andrew Tolmach and Stephanie Weirich and and Steve Zdancewic.

[23] François Pottier and Yann Régis-Gianas. Menhir reference manual. http://gallium.inria.fr/~fpottier/menhir/manual.pdf Version 20151103.

[24] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *CC*, 2010.

[25] Jean Souyris. Industrial use of CompCert on a safety-critical software product. 2004. Presented on 2004-02-04.

[26] WG14/N1256. ISO/IEC 9899:TC3. 2007. http://port70.net/~nsz/c/c99/n1256.pdf Committee draft, September 7.

[27] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.