CS442
Module 10: Systems Programming

University of Waterloo

Winter 2022

"Give a man a program, frustrate him for a day. Teach a man to program, frustrate him for a lifetime."

— Muhammad Waseem

1 Systems Programming

This module focuses on systems programming. We'll discuss the difficulty with modeling systems programming, and introduce CompCert, a formally rigorous C compiler. The goal of this module is to help you think about concrete applications of everything we've done in previous modules, but it will not go into much depth about any of those applications. The formal modeling of systems programming is very much an open field, so this module will mainly present the range of problems, rather than the particular solutions. In essence, we use CompCert as both an exemplar of systems programming and as a success story for using formal semantics to solve real problems, and then look at some other theoretical computer science problems through the lens of systems programming.

2 The Trouble with Systems Programming

Through this entire course, we've focused on formal semantics for reasoning about programming languages. But, whenever we try to relate those formal systems to reality, cracks start to form. What does it mean to get stuck? Just how much junk can I put in σ and Σ? What are labels? What happens with integer overflow, or division by zero?

Usually, formal semantics are used as a tool to prove particular properties. For instance, we may want to prove type safety, or prove that certain programs will always halt. To accomplish this, we eliminate irrelevant language features and focus on just those that interest us. A real language implementation has to handle type errors, but we only have to prove whether they can occur or not, so we don't need to model anything more sophisticated than getting stuck.

As a consequence, formal semantics usually stops well short of systems programming.

"Systems" is a very broad term, and "systems programming" is barely definable as a paradigm. In essence, a language is a systems programming language if it lets us interact with the hardware on a fairly low level. This is ill-defined, because exactly what "low-level interaction" is is unclear, and anyway, most languages could be given low-level access through libraries. And, of course, such access can be very different on different kinds of machines, and one language may be suitable for one kind of machine but not another.

Generally, we restrict the term to languages with a long history of being used to make core systems components, such as operating system kernels, memory managers, and certain language implementations which interact with machine code as both code and data[1]. With such a narrow view, we can reduce the range of potential languages to be considered "systems languages" to a handful: Fortran, Pascal, C, C++, and the assembly languages. There are

[1] Specifically, threaded interpreters, just-in-time compilers, and certain machine-code-macro-based languages such as some implementations of Forth. This does not include standard interpreters or ahead-of-time (i.e., usual) compilers.

some rare exceptions where other languages are used in very special ways, and one upstart which may eventually be added to the list (Rust), but otherwise, systems programming is nearly as sparse a programming paradigm as logic programming.

Our exemplar is C, and more specifically a (slightly) reduced version of C implemented by an unusual compiler, CompCert. Indeed, CompCert C itself is our exemplar; CompCert is a real compiler, but it is also a formally defined programming language! It is assumed that you're already familiar with C, so we won't do our usual introduction to the language syntax.

The minimum unifying feature between all systems programming languages is pointers. We certainly cannot write an operating system kernel or a memory manager without first-class memory addresses. And, that's where systems languages come into conflict with the formally modeling languages as mathematical logic: we have never bothered to actually model memory.

Modeling memory is *hard*. To understand why, consider this C program:

```
int foo(int x) {
    return (&x)[-1];
}
```

This brutish C is perfectly valid C code. In this example it's impossible to know if `foo` actually does anything useful, but similar "out of bounds" behavior isn't at all rare in systems programming. If you've taken CS350 or an equivalent operating systems course, you've undoubtedly written similarly perplexing code. In fact, this code is fairly mundane, since it only *reads* a seemingly out-of-bounds value. We can cause much more mischief like so:

```
int foo(int x) {
    (&x)[-1] = 42;
    return (&x)[-1];
}
```

In these examples, I've also kept things fairly mundane by keeping to data pointers, but certain language implementations routinely cast data pointers to function pointers. How is formal semantics to cope?

A simple answer to this problem is "it's undefined behavior, so we don't have to bother with it". And, if your definition of C is the C specification, that's even right: the C specification leaves the behavior of the above program undefined. But, "undefined behavior" actually has a specific meaning:

> **undefined behavior**
>
> behavior, upon use of a *non-portable* or erroneous program construct or of erroneous data, for which this document imposes no requirements
>
> **Note 1 to entry:** Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to *behaving during translation or program execution in a documented manner characteristic of the environment* (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

— ISO/IEC 9899:2018 (C17 standard) (italics emphasis added) [1]

In particular, note the words which have been highlighted in italics: one correct interpretation of undefined behavior is... documented, and therefore defined, behavior. In fact, undefined behavior as defined by the C standard is simply behavior which is not defined *by the C standard*. In a real system, there are several standards that apply simultaneously, and the C standard is only one of them.

For instance, if you're on a Unix-like system, on the AMD64[2] architecture, then the way memory is laid out, and to a certain extent the way that system calls are performed and the executable format, are defined by the *System V Application Binary Interface AMD64 Architecture Processor Supplement* [3]. With knowledge of how memory is laid out, the correct behavior of the above program can actually be defined. Unfortunately, the above example is still undefined because the memory it accesses is on the stack, the arrangement of which is left to the compiler to define. But, you guessed it, the compiler has a specification too. And, of course, the processor manufacturers publish specifications for their processors. With an exhaustive list of specifications, there is no such thing as undefined

[2] Otherwise known as x86_64 or EM64T. Also known as x64 to people who like inexcusably bad names.

behavior. All programs' behaviors are well-defined, although they may still be non-deterministic or dependent on external factors.

This leaves us with only two problems:

1. All of these specifications are informal, and don't leave room to *prove* anything about systems programming.
2. The implementation of any of the myriad systems between your program and physical reality could have bugs.

The second problem can be solved by the first: if we can *prove* that every layer is correct, by some definition of "correct", and we do so in a way that involves modeling every layer formally, then we can formally reason about the behavior of real programs, and not just theoretical languages with theoretical programs.

> **Aside:** Of course, at the lowest level, the actual physical implementation of a processor comes down to physics, not math. There is always *some* layer at which we must leave the comforting (?) embrace of formal proof, so the goal of formal systems programming is to *minimize* the number of uncertain layers, not to reduce the number to zero.

Formally specifying every layer is a monumental task, but not an impossible one. First we'll discuss the concepts, then we'll discuss our exemplar. As this is a language course, we will focus on everything from the programming language down to the CPU (i.e., not the operating system, and not the physical implementation of the CPU).

3 Formal Models of Memory

Generally, when memory is modeled, it's modeled in a straightforward way: an address-indexed array (or map) of words or bytes. For instance, consider this extremely simple extension to SIL to model memory, with $Heap$ as an address-indexed, word-addressable heap map:

$$\frac{(N_1 \equiv 0) \bmod 4 \qquad Heap' = Heap[N_1 \mapsto N_2]}{\langle \Sigma, \sigma, Heap, {*N_1} = N_2 \rangle \rightarrow \langle \Sigma, \sigma, Heap', \text{skip} \rangle}$$

$$\frac{(N_1 \equiv 0) \bmod 4 \qquad Heap(N_1) = N_2}{\langle \Sigma, \sigma, Heap, {*N_1} \rangle \rightarrow \langle \Sigma, \sigma, Heap, N_2 \rangle}$$

A few things are notable about these rules:

- The word size of our hypothetical architecture (4 bytes) is baked into the system.
- In spite of the above, we haven't even bothered to implement overflow, so our heap isn't particularly accurate.
- We have no memory protection or limits, no consideration for overwriting code or stack with memory access, and no addresses for variables[3].
- `malloc` and `free` would have to be implemented by the user program.

There are at least four major issues that need to be addressed when modeling memory formally:

1. Code is in memory.
2. The stack and variables are in memory.
3. There is a semantic gap between typed memory management (new, or even `malloc`) and raw memory bits.

[3] These restrictions on how memory is used actually match WebAssembly, so they're not unprecedented.

4. Memory has surprising interactions with concurrency.

Generally speaking, the first and second problem are addressed by ignoring them: code is treated as separate, the stack is not (in the formal model) stored in memory, and this is made sufficiently correct by making sure that any memory-accessing code that could try to access the code or stack regions of memory gets stuck.

The semantic gap between `new` and raw memory is surprisingly intricate to address. Consider, for instance, the following (grotesque) C code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      long *a, *b;
6      a = malloc(sizeof(long));
7      b = malloc(sizeof(long));
8      b[-4] = 42;
9      printf("%ld\n", a[0]);
10     return 0;
11 }
```

Whether this code will output `42` depends on precisely how the system's allocator places `a` and `b`. If our formal model models `malloc`, then it *must* lay out memory in some way, and if that way happens to place `a` exactly `4` `sizeof(long)`s below `b`, then in our formal system, this will output `42`. We probably want this code to get stuck, or otherwise be able to identify that its behavior is undefined. But now, consider this code, differing in only one line:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      long *a, *b;
6      a = malloc(sizeof(long));
7      b = malloc(sizeof(long));
8      b[a-b] = 42; // <--
9      printf("%ld\n", a[0]);
10     return 0;
11 }
```

The only difference in this new version is that on line 8, rather than using `b[-4]`, we used `b[a-b]`. This code is now well defined *even by the C standard* (after all, pointer differences are a standard C feature), so it certainly ought to work. If our formal model models `malloc`, and happens to lay out memory as we described above, then `a-b` reduces to `-4`. But, if `a-b` reduces to `-4`, then the actual step we want to *work* in this case is indistinguishable from the step we want to *fail* in the previous case. The only ways to resolve this problem are either (a) employing non-determinism in allocation, and carefully designing our semantics to get stuck if *any* non-deterministic path gets stuck, or (b) putting separate allocations in separate heaps, but creating a new algebra of pointer relationship values by which `a-b` is not simply `-4`, but is a value meaningful only in our semantics that allows us to leap from one heap to another.

Beyond the problem of modeling allocation per se is *type punning*. Type punning is reading memory with a different type than it was written with, intentionally or unintentionally. For instance, a popular technique for implementing dynamic languages on 32-bit systems was[4] *NaN boxing*: values are stored as 64-bit floating-point numbers, but if those numbers are invalid in floating point (Not a Number, or NaN), then the same bits are instead interpreted as a pointer, or 32-bit integer. But, similarly, it's possible to type pun by having two differently typed pointers to the same address, and if this wasn't intended, then the interpretation can be strange (particularly if the target type is itself a pointer!). That "intentionally or unintentionally" is exactly the problem: it's common in C to intentionally perform both simple and complex type punning, and its behavior is well-defined, but platform-specific. A formal model will therefore often get bogged down with a particular platform, modeling not just C, but a particular endianness and even a particular ABI for laying out `struct`s.

Finally, there's the problem with concurrency, which we will discuss (in a very limited fashion) later.

[4] NaN boxing has fallen out of favor, largely because 32-bit systems themselves have fallen out of favor, and it's not worth the effort of maintaining NaN boxing for a dwindling set of systems.

4 Exemplar: CompCert C

CompCert is a C compiler, with an unusual claim to fame: the full compilation process is formally proved correct.

Precisely, to use terminology from *Formal Verification of a Realistic Compiler* [2], this means that if we define S as a source program and C as a compiled program for some specific pair of languages and a compiler between them, and we define $X \Downarrow B$ to mean "executing the program X has observable behavior B", then $S \Downarrow B \implies C \Downarrow B$ for all programs in S which are "safe"[5]. That is, if S compiles to C and S has behavior B, then C has behavior B.

To define this, we need a formal definition for both the language of S and the language of C, and a formal definition of compilation from one to the other. At the extremes, the language of S is C (confusingly; that is, the C programming language), and the language of C is machine code for some architecture. In practice, however, C isn't compiled directly to machine code, but through several compilation phases, and so there are many steps C_1, C_2, $\cdots$, C_n, eventually reaching machine code, and each of these compilation phases must be proved. With a formally defined compilation process, we can simply compose all of these compilation steps to defined compilation all the way from C to machine code, so if the correctness of each of those phases can be proved, then the correctness of the entire compilation process can be proved.

We have a lot of experience formally defining languages in terms of their behavior, but not a lot of experience formally defining compilation. In fact, all the mechanisms are the same, just used for a different purpose: we formally define a compiler by defining a function $Comp$, where $Comp(S) = \text{OK}(C)$, or $Comp(S) = \text{Error}$ if the program cannot compile. That function can be described through (many) formal rules for each specific case, precisely like the $\rightarrow$ morphism.

Thus, to formally prove a compiler, we must do the following:

- Formally define the semantics of a source language, e.g. C.
- Formally define the semantics of a target language, e.g. machine code.
- Formally define the process of compiling, $Comp$.
- Prove that, for any program S in the source language, the behavior of $Comp(S)$ as defined by the formal semantics of the target language is the same as the behavior of S as defined by the formal semantics of the source language.

It is impractical (although not impossible) to write out the formal semantics for a language as large as C, with many non-local effects, without some tool assistance.

Recall that logic programming languages allow us to define *relations*. The $\rightarrow$ morphism is one kind of relation, so we could represent a language semantics in a logic programming language, by defining the clauses for this morphism. If done carefully, a logic programming language could thereby "run" a program in another language, by querying the relation which describes the language's small-step semantics.

In practice, Prolog is not up to the task, because the range of predicates it can prove is quite limited. Instead, CompCert's semantics are implemented in *Coq*, a theorem-proving language with a much more sophisticated theorem-proving algorithm than Prolog's, which allows for a much richer language of predicates, including predicates which are defined themselves as functional programs. The upside of Coq over Prolog is that much more can be proved; the downsides are (a) that while Prolog's algorithm is straightforward and easy to define, so multiple interpreters can implement Prolog, Coq's algorithm is a particular implementation, and subject to change, (b) that Coq's algorithm has unpredictable time bounds, and (c) that almost no anglophone can say "Coq" without snickering.

While we will come nowhere near sharing the entire semantics for C, or converting them into more familiar Post syntax, here is one small example, the (partial) semantics for C addition, as defined by CompCert in Coq, to give you an idea of what Coq definitions look like[6]:

[5] "Safe" is precisely defined for C by CompCert, and essentially means that the program doesn't depend on any behavior that is still undefined after layering the several specifications which go into CompCert.

[6] Source: https://compcert.org/doc/html/compcert.cfrontend.Cop.html

```
Definition sem_add (cenv: composite_env) (v1:val) (t1:type) (v2: val) (t2:type) (m: mem): option val :=
  match classify_add t1 t2 with
  | add_case_pi ty si => (* pointer plus integer *)
      sem_add_ptr_int cenv ty si v1 v2
  | add_case_pl ty => (* pointer plus long *)
      sem_add_ptr_long cenv ty v1 v2
  | add_case_ip si ty => (* integer plus pointer *)
      sem_add_ptr_int cenv ty si v2 v1
  | add_case_lp ty => (* long plus pointer *)
      sem_add_ptr_long cenv ty v2 v1
  | add_default =>
      sem_binarith
        (fun sg n1 n2 => Some(Vint(Int.add n1 n2)))
        (fun sg n1 n2 => Some(Vlong(Int64.add n1 n2)))
        (fun n1 n2 => Some(Vfloat(Float.add n1 n2)))
        (fun n1 n2 => Some(Vsingle(Float32.add n1 n2)))
        v1 t1 v2 t2 m
  end.
```

These semantics are query-able, like Prolog relations, so by defining the semantics, CompCert naturally includes an (absurdly slow) interpreter for C as well.

CompCert includes a semantics for C, for its various intermediate languages, for an idealized but non-specific CPU architecture called "Mach", and for several real CPU architectures: PowerPC, ARM, x86, AMD-64 (x86_64), and RISC-V. Of course, it is always possible that some bug was introduced in the transcription of these languages from their informal specification to the formal specification, or that Coq itself has bugs; such bugs can only be dealt with in the usual way, through years of use and bug hunting.

Proving that an individual compilation step is correct involves proving $S \Downarrow B \implies C \Downarrow B$ for every structure in the language S, inductively for subexpressions.

In similar ways, CompCert defines the semantics for each of its intermediate languages, and for each of its compilation steps.

Note that since CompCert formalizes the particular architecture, and is only really useful when all code (including the memory manager) is analyzed by CompCert, code can have predictable behavior that we would usually want to reject. For instance, our example that uses `b[-4]` to access `a` *may* be considered completely correct, depending on the target and allocator used. As a consequence, the guarantees proffered by CompCert are only guarantees if you actually use the binary produced by CompCert on the architecture it models; it is a verified compiler, *not* a general purpose verification of C. CompCert's generated code isn't very performant, but this is still useful in circumstances where validation is more important than performance, such as systems used in air- and spacecraft.

5 Memory Models Redux

CompCert necessarily models C's pointer-based memory, defining steps for allocating memory, freeing memory, and reading and writing of memory by pointers. The goal in this context is to verify the compiler, and so to verify that C performs the same interaction with memory as S. This model is sufficient for CompCert's goals, but only because CompCert is intended for non-concurrent programs.

Consider the following C snippet:

```
int shared = 0;

void *foo(void *ign) {
    for (int i = 0; i < 1024; i++) {
        shared++;
    }
    return NULL;
}
```

The `foo` function simply increments `shared` 1024 times.

C's implementation of concurrency is shared-memory concurrency. So, if two concurrent threads of execution both run `foo`, they can both read and write to `shared` at the same time. If `foo` is run twice concurrently, what can

be the final value of `shared`?

The answer is that it depends on how the particular CPU implements memory. An `int` is, in all modern systems, 32 bits. Are those 32 bits written all at once? In order from most significant to least significant? One byte at a time? The same question applies to reading. And, of course, incrementing may look like one operation, but it's actually several, including both a memory read and a memory write.

In fact, it's even worse than it sounds. On a modern, multi-core system, if the reader happens to be moved from one core to another, then it's possible for `shared` to end up with a number less than 1024 in the end, as if neither instance of `foo` ran to completion! This is because not all memory accesses go to real memory: there is at least a hierarchy of caches, and they're only guaranteed to be correct for a single thread of execution.

Every CPU that allows true parallelism also has a mechanism (or many mechanisms) for guaranteeing the ordering of two threads' memory accesses, *locking*. But, lock-free concurrent algorithms also exist. To write a lock-free concurrent algorithm requires a deep understanding of exactly what memory orderings are possible in a concurrent program, which is CPU-specific.

Thus, an array of memory models exist, describing the behavior of particular hardware. Broadly, they can be categorized into *strongly consistent* models and *weakly consistent* models, based on whether memory accesses across all concurrent threads appear to happen in a consistent order, or only in a consistent order per each thread. Proving that a particular algorithm works with a particular memory model is of similar complexity to CompCert; creating a single lock-free algorithm is a Ph.D. worth of work.

Unfortunately, there is a trend for languages to simply give up on memory models in their own specifications. C's specification only defines the behavior of programs that use locks to guarantee ordering. Java's assumes strong consistency, so Java implementations must somehow force strong consistency on weakly-consistent architectures.

6 Software Verification

> "It's not a systems language if I can't implement a garbage collector in it."
>
> — Gregor Richards

CompCert's goal is to prove the correctness of a compiler, *not* the compiled program.

Consider, for instance, this snippet of code from a garbage collector, simplified slightly:

```
(struct GGGGC_Pool *) ((size_t) (ptr) & ((size_t) -1 << 24))
```

In short, this takes an address (`ptr`), masks it with `-1 << 24`, and casts the resulting number to a different type of pointer. This strange code is fairly usual for a memory manager.

Consider trying to type-check this code. The type of every expression is clear enough, but could a type-checker prove that the computed address actually points to a `struct GGGGC_Pool`? Although some work has been done on type systems that are sufficient to verify some properties of a memory manager, most type systems are woefully insufficient for interesting systems code.

In fact, type safety is a very weak form of program correctness, but there is a much wider range correctness properties one might want to prove about a program. Usually, these are specific to a particular piece of software or kind of software; for instance, we would want to prove that the above snippet always yields a pointer to a `struct GGGGC_Pool`. More practically, we may want to prove that an elevator is never inoperable when someone is inside, or that a radiation therapy machine never exposes a patient to a dose of radiation above its specification.

This opens us to the area of *software model checking*. Software model checking is (one method for) the verification of particular properties for a piece of software. One can consider type checking to be one very limited form of model checking, and CompCert's proofs of correctness as well, although the model checking community usually defines the term more restrictively.

Programs in safety-critical systems, such as aviation and medical devices, usually use (or at least ought to use!) some combination of software model checking to verify the correctness of their original code, and verified compilation to verify that that code is correctly compiled. Increasingly, international standards bodies actually require this.

7 Just Use Better Languages!

One may reasonably ask why so much focus is put on the verification of systems languages. Surely, if programmers would just use better languages in the first place, these problem wouldn't arise!

There is a degree of truth to this, and it is an unsurprising conceit of many programming language researchers to believe that all software problems can be solved through better languages, but what research there is in the area just doesn't support the idea. Programmers make mistakes with and without type systems, with and without mutation, with and without concurrency.

I have no optimistic conclusion to this topic. Perhaps the lesson to learn from successes such as CompCert is not in using better languages, but in using the right language for the job.

8 Fin

This concludes CS442.

The goal of this course isn't to teach you half a dozen programming languages, although it is to teach you two. The goal isn't even that you use OCaml or Smalltalk after this, although you may want to, of course.

> "A language that doesn't affect the way you think about programming, is not worth knowing."
>
> — Alan Perlis

I hope that exploring the range of programming paradigms has broadened your horizons in the way you think about programming. Just as the goal of this course wasn't to teach you programming languages, the goal of the assignments wasn't to prepare you for implementing languages, since most of you won't. But, being able to think about programming paradigms mechanically allows you to use them as tools in other programming. The power of programming languages is the power of abstraction, and I hope that after having written five language implementations, you feel more comfortable harnessing that power.

References

[1] ISO/IEC 9899:2018: Information technology — Programming languages — C. Standard, International Organization for Standardization, Geneva, Switzerland, June 2018.

[2] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[3] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V application binary interface: AMD64 architecture processor supplement. 2014.

Rights