

An executable semantics for CompCert

Brian Campbell

LFCS, University of Edinburgh



CerCo

Project FP7-ICT-2009-C-243881

February 10, 2012

Introduction

In the **CerCo** project we've been working on
*the construction of a **formally verified** complexity
preserving **compiler** from a large subset of **C** to some
typical microcontroller assembly*

Inspired by (and borrowing a little from) Leroy et al's **CompCert**.

They define languages by **small-step inductive** definitions.
We define language with **executable** interpreters.

Executable semantics are easier to test.

Can we retrofit executable semantics to CompCert and find out
anything interesting?

C is quirky, flawed, and an enormous success.

— dmr, HOPL'93.

What's so difficult about C?

Around 160 A4 pages of specification (400 with libraries added).

What's so difficult about C?

Around 160 A4 pages of specification (400 with libraries added).

Implicit conversions:

```
int x = 'a' + 0.5;
```

What's so difficult about C?

Around 160 A4 pages of specification (400 with libraries added).

Implicit conversions:

```
int x = 'a' + 0.5;
```

Mixed reads and writes of an object are undefined:

```
x = i + i++;
```

What's so difficult about C?

Around 160 A4 pages of specification (400 with libraries added).

Implicit conversions:

```
int x = 'a' + 0.5;
```

Mixed reads and writes of an object are undefined:

```
x = i + i++;
```

Evaluation order constraints very lax, not uniform:

```
x = i++ && i++;
```

```
x = i++ & i++;
```

What's so difficult about C?

Around 160 A4 pages of specification (400 with libraries added).

Implicit conversions:

```
int x = 'a' + 0.5;
```

Mixed reads and writes of an object are undefined:

```
x = i + i++;
```

Evaluation order constraints very lax, not uniform:

```
x = i++ && i++;  
x = i++ & i++;
```

Annoying corner cases:

```
int x[];  
int main() { return x[0]; }
```


CompCert History (up to 1.8)

- ▶ CompCert starts with **big-step Clight** semantics
 - ▶ Side-effect free expressions, no gotos.
 - ▷ Some of the literature refers to this version.

CompCert History (up to 1.8)

- ▶ CompCert starts with **big-step Clight** semantics
 - ▶ Side-effect free expressions, no gotos.
 - ▷ Some of the literature refers to this version.
- ▶ Switch to **small-step Clight** semantics
 - ▶ Side-effect free expressions, gotos.
 - ▷ CerCo project started from here

CompCert History (up to 1.8)

- ▶ CompCert starts with **big-step Clight** semantics
 - ▶ Side-effect free expressions, no gotos.
 - ▷ Some of the literature refers to this version.
- ▶ Switch to **small-step Clight** semantics
 - ▶ Side-effect free expressions, gotos.
 - ▷ CerCo project started from here
- ▶ Small-step **CompCert C** language
 - ▶ C-like expressions,
 - ▶ gotos, and ...

CompCert History (up to 1.8)

- ▶ CompCert starts with **big-step Clight** semantics
 - ▶ Side-effect free expressions, no gotos.
 - ▷ Some of the literature refers to this version.
- ▶ Switch to **small-step Clight** semantics
 - ▶ Side-effect free expressions, gotos.
 - ▷ CerCo project started from here
- ▶ Small-step **CompCert C** language
 - ▶ C-like expressions,
 - ▶ gotos, and ...

The latter comes in two flavours:

1. A **non-deterministic** version (the intended input language)
2. A **deterministic** version (what the compiler actually does)

CompCert and testing

Untrustworthy OCaml Formal development in Coq

$C \rightarrow \text{CompCert } C \rightarrow \text{Clight} \xrightarrow{\text{formal proof}} \text{ASM} \rightarrow \text{Machine code}$

Coq sections get 'extracted' to OCaml for execution.

There's a formal proof in the middle,
but the edges are a bit worrying.

CompCert and testing

Untrustworthy OCaml Formal development in Coq

$C \rightarrow \text{CompCert } C \rightarrow \text{Clight} \xrightarrow{\text{loop}} \text{ASM} \rightarrow \text{Machine code}$

$C \rightarrow \quad \rightarrow \quad \xrightarrow{\text{loop}} \quad \rightarrow \text{Machine code}$

Normal testing tries all of the code.

CompCert and testing

Untrustworthy OCaml Formal development in Coq

$C \rightarrow \text{CompCert } C \rightarrow \text{Clight} \xrightarrow{\text{proof}} \text{ASM} \rightarrow \text{Machine code}$

$\text{CompCert } C \rightarrow \text{Clight} \xrightarrow{\text{proof}} \text{ASM}$

Proofs exercise the formal development.

- ▶ Tactical interactive theorem proving helps you notice bad definitions

CompCert and testing

Untrustworthy OCaml Formal development in Coq

$C \rightarrow \text{CompCert } C \rightarrow \text{Clight} \xrightarrow{\text{infinite loop}} \text{ASM} \rightarrow \text{Machine code}$

$C \rightarrow \text{CompCert } C$

With an executable semantics we can test the first part.

- ▶ Holes in the specification can mask holes in the proof
- ▶ Also get to play 'spot the undefined behaviour' game

Constructing the executable semantics

CompCert provides us with a head start:

- ▶ the **memory model** is executable,
- ▶ local and global **environments** are defined in terms of functions,
- ▶ the semantics of **operators** such as $+$, $==$, etc are defined by functions,
- ▶ an **error monad** is available for failing.

In particular, environments are used by the compiler, so they are also fairly efficient.

Constructing the executable semantics

Syntax directed relations are easy to make functions from:

```
Inductive lred: expr -> mem -> expr -> mem -> Prop :=  
  | red_var_local: forall x ty m b,  
    e!x = Some(b, ty) ->  
    lred (Evar x ty) m  
      (Eloc b Int.zero ty) m  
  ...
```

Constructing the executable semantics

Syntax directed relations are easy to make functions from:

```
Inductive lred: expr -> mem -> expr -> mem -> Prop :=  
  | red_var_local: forall x ty m b,  
    e!x = Some(b, ty) ->  
    lred (Evar x ty) m  
      (Eloc b Int.zero ty) m  
  ...
```

```
Definition exec_lred (e:expr) (m:mem) : res (expr * mem) :=  
match e with  
| Evar x ty =>  
  match en!x with  
  | Some (b, ty') => match type_eq ty ty' with  
    | left _ => OK (Eloc b Int.zero ty, m)  
    | right _ => Error (msg "type mismatch")  
  end  
  ...
```

Constructing the executable semantics — non-determinism

We encode strategies as functions

```
expr -> kind * expr * (expr -> expr)
```

and require that it really does give a subexpression and context.

Doesn't cover all strategies:

- ▶ Implementations could use contextual information, randomness. . .
- ▶ various methods can solve this, but not terribly important here

Constructing the executable semantics — stuck subexpressions

The non-deterministic semantics check for stuck subexpressions.

- ▶ picks up non-terminating programs with undefined behaviour
- ▶ example where `f` does not terminate:

```
f() + (10 / x)      with x = 0
```

- ▶ should be able to get stuck after substituting `x`
- ▶ but without check we can always reduce `f()`

Naïve implementation would be inefficient:

any subexpression in an evaluation context is either a value, or has a further subexpression that is reducible

but there is a nice structurally recursive version.

Soundness and completeness

We want to know that the executable semantics does the same thing as the original semantics.

- ▶ (mostly boring) inductive proofs
- ▶ Coq's `Function` feature for generating induction principles tailored to particular functions is great, but still a bit limited

Caveats apply to completeness:

- ▶ Limitations on strategies — cheat by single-stepping
- ▶ No I/O (CerCo uses a resumption monad for I/O.)

Strategies and the deterministic semantics

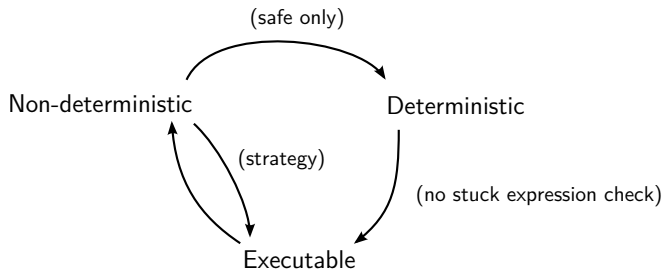
Two variants have been implemented:

1. a simple left-most inner-most strategy,
2. the actual strategy implemented by the compiler

Strategies and the deterministic semantics

Two variants have been implemented:

1. a simple left-most inner-most strategy,
2. the actual strategy implemented by the compiler



Completeness proof interesting:

- ▶ Deterministic semantics has big-step for 'simple' expressions
- ▶ Proof shows that this really does correspond to non-deterministic

OCaml driver code

Complete the interpreter with some untrustworthy OCaml:

1. Repeat the Coq step function until the program stops or fails.
2. Add optional code to work around bugs
 - ▶ don't need to fix them properly
 - ▶ don't need to prove anything
3. Also good for hacks: `memcpy`, `printf`, ...
Implement things outside of CompCert's model of C.

Testing — function pointers

The example that I originally wanted to try.

```
int zero(void) { return 0; }

int main(void) {
    int (*f)(void) = zero;
    return f();
}
```

```
$ ../compcert-git-badfn/cexec fnptr-simple.c
stuck expression:  function value hasn't a function type
```

The function call rule requires `f` to evaluate directly to a function, not a pointer.

Testing — function pointers

The example that I originally wanted to try.

```
int zero(void) { return 0; }

int main(void) {
    int (*f)(void) = zero;
    return f();
}
```

Fixing this is easy — the compiler **already** had the correct type check!

And the proof scripts got shorter.

Testing — Csmith

Random program generator by Yang et al from U. Utah.

- ▶ Targets 'middle-end' bugs
- ▶ Regular testing only found bugs in untrustworthy OCaml code
- ▶ Random code didn't find any errors in semantics

Testing — Csmith

Random program generator by Yang et al from U. Utah.

- ▶ Targets ‘middle-end’ bugs
- ▶ Regular testing only found bugs in untrustworthy OCaml code
- ▶ Random code didn’t find any errors in semantics

...but the **non-random** code of safe mathematics functions. . .

Testing — Csmith

...but the **non-random** code of safe mathematics functions...

```
int8_t lshift_func_int8_t_s_s(int8_t left, int right)
{
    return
        ((left < 0) ||
         (((int)right) < 0) ||
         (((int)right) >= 32) ||
         (left > (INT8_MAX >> ((int)right)))) ?

    left :

    (left << (((int)right)));
}
```

Semantics is missing arithmetic conversion for ?;.

But the compiler works on this example, because 'all' integers are 32 bits.

Testing — Csmith

Semantics is missing arithmetic conversion for ?;.

But the compiler worked on that example, because 'all' integers are 32 bits.

```
double f(int x, int a, double b) {  
    return x ? a : b;  
}
```

Testing — Csmith

Semantics is missing arithmetic conversion for ?;.

But the compiler worked on that example, because ‘all’ integers are 32 bits.

```
double f(int x, int a, double b) {  
    return x ? a : b;  
}
```

The compiler is missing the conversion too:

```
$ ../compcert-git/ccomp conditional.c  
Error during RTL type inference:  type mismatch  
In function main:  RTL type inference error
```


Testing — Csmith

Semantics is missing arithmetic conversion for ?;.

But the compiler worked on that example, because ‘all’ integers are 32 bits.

```
double f(int x, int a, double b) {  
    return x ? a : b;  
}
```

The compiler is missing the conversion too:

```
$ ../compcert-git/ccomp conditional.c  
Error during RTL type inference:  type mismatch  
In function main:  RTL type inference error
```

We made a failing test-case from a working one.

Testing — gcc-torture

An executable subset of GCC's C test suite, pre-filtered by another executable semantics project (kcc from U. Illinois).

Lots of fun:

- ▶ lack of initialisation
 1. only in the semantics, and
 2. not in the compiler in OCaml
- ▶ a little array/pointer confusion (OCaml)
- ▶ incomplete array type mismatches (both, kind of)
- ▶ Missing trivial cases for cast (semantics, fixed already)
- ▶ pointer comparisons (semantics, intentional limitation)
- ▶ bad line numbers in errors (OCaml)
 - ▶ not helped by OCaml's non-deterministic evaluation order...

Related work

CompCert response

- ▶ bugs fixed, sometimes before I found them
- ▶ fresh interpreter implementation
 - ★ inspired by this work, but different: finds all possible redexes, turns out smaller and neater; doesn't explicitly do deterministic semantics

Lots of other executable semantics exist

- ▶ kcc, CompCertTSO, some JVMs, ...
- ▶ often the natural way to use a system (e.g., ACL2)
Milner and Weyhrauch 1972

More fun things you can do

- ▶ Add I/O, full program evaluation
- ▶ Check for coverage

Conclusions

Took an existing verified compiler,

- ▶ added an executable version of the semantics,
- ▶ found bugs through testing,
 - ★ including a bug in the formalized front-end
 - ★ even though the original test-case is compiled properly
- ▶ useful for illustrating limitations of the semantics, especially ones you didn't know about,
- ▶ showed that the semantics cope with a large group of tests,
- ▶ showed a connection between the original deterministic and non-deterministic semantics.