An Executable Semantics for CompCert C*

Brian Campbell

LFCS, University of Edinburgh,
Brian.Campbell@ed.ac.uk

Abstract. CompCert is a C compiler developed by Leroy et al, the majority of which is formalised and verified in the Coq proof assistant. The correctness theorem is defined in terms of a semantics for the 'CompCert C' language, but how can we gain faith in those semantics? We explore one approach: building an equivalent executable semantics that we can check test suites of code against.

Flaws in a compiler are often reflected in the output they produce: buggy compilers produce buggy code. Moreover, bugs can evade testing when they only manifest themselves on source code of a particular shape, when particular optimisations are used. This has made compilers an appealing target for mechanized verification, from early work such as Milner and Weyhrauch's simple compiler in LCF [15] to modern work on practical compilers, such as the Verisoft C0 compiler [10] which is used as a key component of a larger verified software system.

The CompCert project [11] has become a nexus of activity in recent years, including the project members' own efforts to refine the compiler and add certified optimisations [22,18,8], and external projects such as an extensible optimisation framework [20] and compiling concurrent programs [23]. The main result of the project is a C compiler which targets the assembly languages for a number of popular processor architectures. Most of the compiler is formalised in the Coq proof assistant and accompanied by correctness results, most notably proving that any behaviour given to the assembly output must be an acceptable behaviour for the C source program.

To state these correctness properties requires giving some semantics to the source and target languages. In CompCert these are given by inductively defined small-step relations. Errors and omissions in these semantics can weaken the overall theorems, potentially masking bugs in the compiler. In particular, if no semantics is given to a legitimate C program then any behaviour is acceptable for the generated assembly, and the compiler is free to generate bad code[1]. This corresponds to the C standard's notion of *undefined behaviour*, but this notion is left implicit in the semantics, so there is a danger of underdefining C.

* The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881

[1] By design, the compiler is also free to fail with an error for *any* program, even if it is well-defined.

Several methods could be used to gain faith in the semantics; in the conclusions of [11] Leroy suggests manual review, testing of executable versions and proving connections to alternative forms of semantics. Here we investigate testing an executable semantics which, by construction, is closely and formally related to CompCert's input language. The testing is facilitated by Coq's extraction mechanism, which produces OCaml code corresponding to the executable semantics that can be integrated with CompCert's existing parser to produce a C interpreter.

Our motivation for this work comes principally from the CerCo project where we use executable semantics throughout [1], and in particular have an executable version of CompCert's Clight intermediate language [5]. There we found an omission involving function pointers during testing. We wished to attempt to replicate this in CompCert C and search for any other evident problems. An executable semantics is far more practical for this task than manual animation of the inductive definitions[2]. Morever, CompCert C is a more complex and interesting language than Clight, especially due to the presence of side-effects and non-determinism in the evaluation of expressions.

The main contributions of this paper are

1. demonstrating the successful retrofitting of a small-step executable semantics to an existing verified compiler with equivalence proofs against the original semantics,
2. showing that testing of the resulting interpreter does lead to the identification of bugs in both the semantics and the compiler,
3. showing that the executable semantics can illustrate the limitations of the semantics, and fill in a gap in the relationship between the original deterministic and non-deterministic versions of the semantics, and
4. demonstrate that a mixture of formal Coq and informal OCaml code can make testing more effective by working around known bugs with little effort.

The full development is available online as a modified version of CompCert[3].

In Section 1 we will give an overview of the relevant parts of the CompCert compiler. In Section 2 we will discuss the construction of the executable semantics, its relationship to the inductively defined semantics from CompCert (both formalised in Coq) and the additional OCaml code used to support testing. This is followed by the results of testing the semantics in Section 3, including descriptions of the problems encountered. Finally, Section 4 discusses related work, including an interpreter Leroy added to newer versions of CompCert following this work.

[2] This is partly because manual animation requires providing witnesses for the results or careful management of existential metavariables, and partly because we can prove results once for the executable semantics rather than once per program. Automatic proof search has similar problems.

[3] `http://homepages.inf.ed.ac.uk/bcampbe2/compcert/`

1 Overview of the CompCert compiler

CompCert compiles a simplified but substantial subset of the C programming language to one of three different architectures (as of version 1.8.2, which we refer to throughout unless stated otherwise). The main stages of the compiler, from an abstract syntax tree of the 'CompCert C' language to target specific pseudo-assembly code, are written in the Calculus of Inductive Constructions (CIC) — the dependently-typed language that underlies the Coq proof assistant [21]. The formal proofs of correctness refer to these definitions. Coq's extraction facilities [13] produce OCaml code corresponding to them, which can be combined with a C parser and assembler text generation to complete the compiler. The resulting compilation chain is summarised in Figure 1.

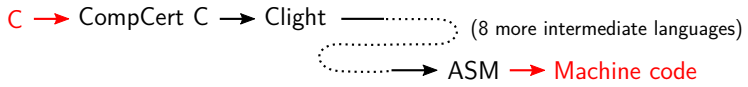

Fig. 1. Start and end of the CompCert compilation chain

Note that the unformalised C parser is not a trivial piece of code. Originally based on the CIL library for C parsing [17] but heavily adapted, it not only produces an abstract syntax tree, but also calculates type information and can perform several transformations to provide partial support for features that are not present in the formalised compiler. In particular, bitfields, structure passing and assignment, and `long double` and `long long` types are (optionally) handled here. There is a further transformation to simplify the annotated C abstract syntax tree into the CompCert C language.

Earlier versions of CompCert did not feature CompCert C at all, but the less complex Clight language. Clight lives on as the next intermediate language in the compiler. Hence several publications on CompCert refer to this language [3]. The major difference between the languages is that the syntax and semantics of expressions in Clight are much simpler because they are deterministic and side-effect free. Moreover, CompCert's original semantics were in big-step form and lacked support for `goto` statements. Version 1.5 added these using a small-step semantics, and 1.8 added the CompCert C language with its C-like expressions. The latter effectively moved some of the work done by the OCaml parser into the formalised compiler.

CompCert C is also interesting because it has two semantics: one with non-deterministic behaviour for expressions which models the uncertainty about evaluation order in C, and a deterministic semantics which resolves this uncertainty. The former can be regarded as the intended input language of the compiler, and the latter as a more specific version that the compiler actually implements. In addition, these semantics have slightly different forms: in particular the non-deterministic version is always small-step, whereas the deterministic form makes

a single big-step for side-effect free subexpressions. The two are connected by a result showing that programs which are *safe* in the non-deterministic semantics admit some execution in the deterministic semantics, where by *safe* we mean that the program never gets stuck regardless of the order of evaluation used (for a fixed choice of I/O behaviour). This result essentially holds because the deterministic semantics is choosing one of the possible evaluation orders.

Note that throughout the definitions of all of these semantics we benefit from a specialisation of C to the targets that CompCert supports. Various *implementation-defined* parts of the language such as integer representation and sizes are fixed (in fact, the formalisation heavily uses the assumption of a 32-bit word size). Also, some *undefined behaviour* is given a meaning; notably mixtures of reads and writes to a variable in a single expression are given a non-deterministic behaviour rather than ruled out as per the C standard [4, §6.5.16].

2 Construction of the executable semantics

The executable semantics consists of several parts: existing executable definitions from CompCert, functions which closely correspond to the original relational semantics, a mechanism for resolving non-deterministic behaviour, and finally the proofs that steps of the relational semantics are equivalent to steps of the executable semantics. We do not consider whole program executions or interaction with the outside world (I/O) here, but only individual steps of the semantics. It is not difficult to include these (we have done this for Clight as part of the CerCo project [5]), but we do not expect that it would yield any worthwhile insights during testing to justify the effort involved arising from the extra non-determinism from I/O and the coinductive reasoning for non-terminating programs.

The existing definitions that we can reuse include the memory model that is used throughout the CompCert development [12]. This model features symbolic pointers with discrete abstract blocks of memory, and so is more suitable for our purposes than non-executable alternatives such as a concrete model with non-deterministic choice of fresh locations for allocations. The semantics of operations on values (for example, `+` and `==`) are naturally defined as auxiliary functions in the relational semantics, which we reuse. Casts and the conversion of values to booleans are exceptions to this; they are defined inductively in the relational semantics and we treat them in the same way as the rest of the inductive definitions, below.

Local and global variable environments are necessarily executable in CompCert because they are used throughout the actual compiler code in addition to the semantics. We also make use of the compiler's error monad to recover the partiality of the relational semantics (that is, CIC is a language of total functions, so we must model failure explicitly).

The environments are efficiently implemented because of their use in the compiler itself. However, the memory model was not intended to be efficient and builds large chains of closures to represent changes to memory. Fortunately, the performance was sufficient for our testing, so we did not attempt to optimise it.

2.1 Executing a step of the semantics

The relational semantics uses a series of inductive definitions to describe the steps of the abstract machine: one for reducing *lvalues*, expressions which represent a location that can be read from, written to, or extracted as a pointer; one for *rvalues* expressions that only yield a value, such as addition; and one for setting up function calls:

```
Inductive lred: expr -> mem -> expr -> mem -> Prop := ...
Inductive rred: expr -> mem -> expr -> mem -> Prop := ...
Inductive callred: expr -> fundef -> list val -> type -> Prop := ...
```

The `estep` relation represents whole state transitions corresponding to the reductions on expressions given by the first three relations, and the `sstep` relation performs steps of statements and function entry and exit:

```
Inductive estep: state -> trace -> state -> Prop := ...
Inductive sstep: state -> trace -> state -> Prop := ...
```

The union of these two relations gives the overall `step` relation.

The executable semantics uses functions corresponding to each of these relations. The syntax directed nature of the inductive definitions makes this task straightforward. For example, consider the `rred` rules for conditional expressions:

```
| red_condition_true: forall v1 ty1 r1 r2 ty m,
    is_true v1 ty1 -> typeof r1 = ty ->
    rred (Econdition (Eval v1 ty1) r1 r2 ty) m
         (Eparen r1 ty) m
| red_condition_false: forall v1 ty1 r1 r2 ty m,
    is_false v1 ty1 -> typeof r2 = ty ->
    rred (Econdition (Eval v1 ty1) r1 r2 ty) m
         (Eparen r2 ty) m
```

These state that a true conditional reduces to its left subexpression and a false one to its right, subject to a typing constraint.

We rearrange the rules to fit in a tree of pattern matching, check type equalities where necessary, and introduce error messages where no reduction is possible:

```
Definition exec_rred (e:expr) (m:mem) : res (expr * mem) :=
match e with
...
| Econdition (Eval v1 ty1) r1 r2 ty =>
    do b <- exec_bool_val v1 ty1;
    let r := if b then r1 else r2 in
    match type_eq (typeof r) ty with
    | left _ => OK (Eparen r ty, m)
    | right _ => Error (msg "type mismatch in Econdition")
    end
```

This transforms the value `v1` to a boolean, selects the appropriate subexpression and checks its type. The '`do`' notation is a use of the error monad: the conversion to a boolean is not always defined and may return an error, which the

monad propogates to the caller of `exec_rred`. The auxiliary relations defining the relationship between values and booleans, `exec_bool_val`, and casting are also defined as functions.

Defining a function for the statement reduction, `sstep`, is similar. One part of function entry handling deserves particular attention. CompCert has a notion of *external functions*, which are a set of primitive CompCert C functions that are handled outside of the program to provide system calls, dynamic memory allocation, volatile memory accesses and annotations. Inductive predicates describe what changes to the program's state are allowed. We have only implemented the dynamic memory allocation (by constructing a function similar to the predicates for `malloc` and `free`), and provide a predicate on states to identify uses of the other calls so that we may give a partial completeness result in the next section.

The expression relation `estep` is more difficult. This is the point in the semantics at which non-deterministic behaviour appears — any subexpression in a suitable *context* can be reduced. The contexts are defined by an inductive description of which subexpressions can be evaluated in the current state (for example, only the guard of a conditional expression can be evaluated before the conditional expression itself, whereas either subexpression of e_1 `+` e_2 can be reduced). We could execute the non-deterministic semantics by collecting all of the possible reductions in a set (in the form of the possible successor states), but we are primarily interested in testing the semantics on well behaved C programs, and an existing theorem in CompCert shows that any program with a *safe* non-deterministic behaviour also has a behaviour in the deterministic semantics.

Hence we are satisfied with following a single path of execution, and so parametrise the executable semantics by a *strategy* function which determines the evaluation order by picking a particular subexpression, and also returns its context. These contexts should be valid with respect to the inductive definition mentioned above. Moreover, if we choose a function matching the deterministic semantics then we know that any well-defined C program which goes wrong is demonstrating a bug (in the form of missing behaviour) which affects *both* sets of semantics. Regardless of the choice of strategy, we can used failed executions to pinpoint rules in the semantics which concern us. We will discuss the implemented strategies and their implications in the following section.

Finally, to match the non-deterministic semantics of `estep` precisely we must also check that there are no *stuck* subexpressions. This detects subexpressions which cannot be reduced because they are erroneous, rather than because they are already values. Normally the lack of any reducible subexpression would indicate an error, but the extra check is required because an accompanying non-terminating subexpression will mask it. The CompCert documentation gives the example `f() + (10 / x)` where `x` is initially 0 and `f` is non-terminating. Without the check, a program containing this expression would appear to be well-defined because `f` can always be reduced, whereas we should make the program undefined because of the potential division by zero.

Informally, the condition for `estep` states that *any subexpression in an evaluation context is either a value, or has a further subexpression that is reducible.*

This form suggests an inefficient executable approach: for every context perform a search for the reducible subexpression. Fortunately it can be implemented by a relatively direct recursive function because we can reuse reducible subexpressions that we have already found in a larger context. Thus we only have to check that an expression can be reduced (using the reduction functions discussed above) when all of the subexpressions have already been reduced to values.

The deterministic semantics does not enforce this condition because it commits to one particular evaluation, regardless of whether other evaluations may fail. This means that the relationship between the non-deterministic and deterministic semantics is not a straightforward inclusion: the deterministic semantics accepts some executions where the stuckness check fails, but rejects the executions allowed by the non-deterministic semantics under a different expression reduction order. Thus we make the executable version of the check optional so that we may implement either semantics.

2.2 Equivalence to the non-deterministic semantics

We show the equivalence of the executable semantics to the original relational semantics in two parts, summarising the proofs from the formal Coq development. First, we show that any successful execution of a step corresponds to some derivation in the original semantics, regardless of the strategy we choose.

Theorem 1 (Soundness). *Given a strategy which picks valid contexts, the execution of any step s_1 to s_2 with trace t implies that there exists a derivation of* `step` s_1 t s_2.

The proof consists of building similar lemmas for each relation in the semantics, using case analysis on the states, statements and expressions involved, reduction of the executable hypothesis and reasoning on equalities until the premises of the constructor are realised. The check for *stuck* subexpressions is the most involved task: we must prove a more general result differentiating between reducible and fully reduced expressions, and show that reducible terms found in subexpressions can be lifted. If no such term exists, we show that attempting to reduce the whole term is sufficient.

The second part is to show that any step of the original semantics is successfully executed by the executable semantics.

Theorem 2 (Completeness — non-deterministic). *For any derivation of* `step` s_1 t s_2 *which is not about to call an unsupported external function there exists a strategy p such that* `exec_step` p s_1 $=$ `OK` (t, s_2).

As in the soundness proof, a lemma is shown for each relation. The general form of the proofs is to perform inversion on the hypothesis for the derivation and use rewriting and the prior lemmas to reduce the goal. Again, the stuckness check is more difficult: it follows by induction on the expression, and showing that if a subexpression gets stuck then the parent expression gets stuck too.

Note that each step might require a different strategy because we have restricted our attention to strategies that can be expressed as functions on the abstract machine state. This rules out strategies where the execution order depends on (for example) previous states or randomness, but these do not occur in the CompCert compiler.

2.3 Strategies and the deterministic semantics

Two evaluation strategies were implemented for the executable semantics. The first was a simple leftmost-innermost strategy that picks the first non-value subexpression that we are allowed to reduce. It was chosen because it is simple to implement, sufficient for testing (any bugs were likely to be independent of evaluation order, and our experience with the second strategy supported this), and could be used as the basis for the second strategy.

However, to provide the greatest benefit from the executable semantics we wanted to get executions which precisely match the deterministic semantics that the compiler actually implements. That is, we wish to have an interpreter that predicts exactly the behaviour of the compiled program. This is not entirely straightforward because the original deterministic semantics are formalised in a slightly different way to non-deterministic executions: expressions with no side effects are dealt with by a big-step relation, and effectful expressions use a small-step relation as before. Here 'effectful' includes both changes to the state and to control flow such as the conditional operator. This prioritisation of the effectful subexpressions is also what causes the difference in evaluation order. For example, `x+(x=1)` when `x` $\neq 1$ gives a different answer with the second strategy because the assignment is evaluated first.

Despite the difference in construction, we can still encode the resulting evaluation order as a strategy:

1. find the leftmost-innermost *effectful* subexpression (or take the whole expression if it is effect-free), then
2. (a) pick the leftmost-innermost *effect-free* non-value subexpression of that if one is present, otherwise
   (b) reduce the whole subexpression.

Intuitively, this works because the effect-free subexpressions can be reduced in any order that is allowed without changing the result.

To show formally that this matches the relational semantics we first show that the big-steps of effect-free subexpressions can be decomposed into a series of small-steps using the relations from the non-deterministic semantics. We can then show that iterating the executable semantics will perform the same steps, reaching the same end state as the big-step. Then using some lemmas to show that the `leftcontext`s (the deterministic version of expression contexts) used in the semantics are found by our leftmost-innermost effectful subexpression strategy, we can show that any step of the deterministic semantics is performed by some number of steps of the executable semantics:

Theorem 3 (Completeness — deterministic). *For any derivation of* `step` s_1 t s_2 *in the deterministic semantics which is not about to call an unsupported external function, repeated use of the* `exec_step` *function starting with* s_1 *using the above strategy yields the state* s_2*. Moreover, the concatenation of the resulting traces is* t*.*

Again, a precise version of the above argument is formalised in Coq.

Together with the connections between the non-deterministic semantics and the executable semantics, and the result from CompCert linking the two relational semantics we get the relationships depicted in Figure 2. We can see that

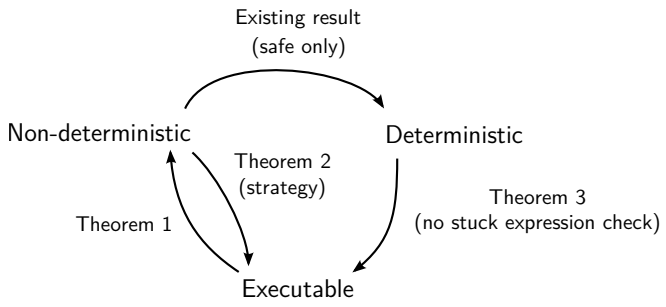

Fig. 2. Relationships between the different CompCert C semantics

the new result completes a loop — confirming the intuition that the deterministic semantics is the same as the non-deterministic semantics with a particular choice of evaluation order, but without the 'stuckness-check'.

We also see directly from the executable semantics that the evaluation order is the *only* possible source of non-determinism besides I/O, because each step of the semantics can be expressed as a function rather than a relation.

2.4 Informal OCaml code

To produce the actual interpreter we add some OCaml code which uses the existing CompCert parser to produce CompCert C from C source code, and then iterates the step function extracted from Coq until the program successfully completes, or fails with an error.

Ideally we would fix each bug that we encounter when testing this interpreter, but this can involve repairing the specifications, the compiler, and the proofs; all of which CompCert's developers are better placed to deal with. Nonetheless, we wish to continue testing in spite of any bugs found. To achieve this we detect troublesome states in the OCaml main loop and override the normal behaviour of the executable semantics without changing the formal development. To assist in the implementation of these workarounds we use a higher order function to

apply a local change to an expression to any applicable subexpression appearing in an evaluation context.

We also take the opportunity to add some support functions that are outside the scope of the semantics, but which are informally supported by the compiler. They are implemented by detecting states which call these functions and running an OCaml version instead. We provide `exit` and `abort` (trivial functions to halt execution), `memcpy` and `memcmp` (impossible to implement in CompCert C because the memory model does not provide byte-by-byte representations of pointers[4]) and a limited version of `printf`. CompCert's semantics also assume that the entire program is present in a single file, so we provide some support for merging definitions from multiple files to simulate separate compilation.

All of these can be added without requiring any extra proof or changing the compiler or the semantics. They can also be turned off to be sure that problems we encounter come from the formalised semantics and not the workarounds.

Finally, we note that the execution of the semantics can be examined in the debugger packaged with OCaml when absolute certainty about the cause of a problem is required. Indeed, during the testing below the debugger was used to retrieve extra information about the program state that was not included in the interpreter's error message.

3 Testing

After some basic testing to ensure that the interpreter was functioning correctly (in particular, testing the informal driver code), we proceeded with the example that illustrated an omission in CerCo's Clight semantics, then attempted more demanding tests.

3.1 Function pointers

The simple program

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

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

returns zero by a call to the `zero` function via a function pointer. It failed during testing of CerCo's executable semantics due to a bad type check, and CompCert C was suspected to suffer from the same problem. Indeed, execution fails at the call with the same type error: the variable `f`'s type is a pointer to a function and the semantics only accepts a function type.

[4] Implementation of `memcpy` and `memcmp` was assisted by the fact that Coq's extraction process exposes the internal representation of the memory model, rather than forcing us to implement them through the model's interface.

There are two noteworthy aspects to this bug. First, the compiler itself features the correct type check and works perfectly. The error only affects the specification of the compiler; this is extra behaviour that has not been proved correct. The semantics can be easily modified to use the same check, and the only other change required is to *remove* some steps from the proof scripts. Second, the bug only became relevant from version 1.7 of CompCert — before which the parser always added an explicit dereference. This illustrates a general issue: the intended semantics of CompCert C is unclear because it is difficult to understand what we should assume about the output of the complex parser.

3.2 Csmith

Csmith is a tool for the random generation of test cases for C compilers that uses a mixture of static and dynamic checks to ensure that the generated code has well-defined behaviour [24]. Failing test cases are identified by the compiler failing with a crash or error, the generated code crashing, or by comparing the output of the test case with that obtained using different compilers. Each generated test case outputs a checksum to summarise its runtime behaviour for comparison.

Csmith is particularly interesting because it has already been used to test the CompCert compiler as a whole where it detected several bugs in the informal part of the compiler and a mismatch between the assembler output and the assembler's behaviour. These problems have since been corrected, partly by the introduction of CompCert C as the input language. Thus it gives us the opportunity to compare the semantics against the actual compiler. However, it is not an ideal tool for testing semantics because it focuses on detecting 'middle-end' bugs, and thus only generates a limited range of front-end language features.

Given the previous testing of the compiler, it was unsurprising that we found no problems when executing the randomly generated code with the interpreter and comparing the results against the compiler[5]. However, we did encounter a failure with the *non-random* support code which implements the dynamic checks that prevent undefined behaviour in arithmetic operations. The failing code can be seen in Figure 3.

The failure is caused by the reduction rules for conditional expressions:

```
| red_condition_true: forall v1 ty1 r1 r2 ty m,
    is_true v1 ty1 -> typeof r1 = ty ->
    rred (Econdition (Eval v1 ty1) r1 r2 ty) m
         (Eparen r1 ty) m
| red_condition_false: forall v1 ty1 r1 r2 ty m,
    is_false v1 ty1 -> typeof r2 = ty ->
    rred (Econdition (Eval v1 ty1) r1 r2 ty) m
         (Eparen r2 ty) m
```

Each rule requires the type of the chosen subexpression to equal the type of the entire expression. However, here one branch is an 8-bit integer, and the other a 32-bit integer so one of the rules cannot be applied. The C standard requires

[5] Using Csmith version 2.0.

```
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));
}
```

Fig. 3. Excerpt of the safe arithmetic code from Csmith (edited for readability)

that 'the usual arithmetic conversions' should be applied to coerce the result to a common type [4, §6.5.15], so the 8-bit integer should be promoted to a 32-bit one.

As with the function pointers, this test works with the compiler because the two types have a common representation — all integers are handled as 32-bit words in CompCert. It is only when they are loaded and stored to memory that the integer size is taken into account. Nonetheless, this failure provides us with enough information to construct an example on which the compiler fails because the representations differ:

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

Fortunately, the compiler performs some type reconstruction for the RTL intermediate language which makes it produce an error rather than bad code. Curiously, this means that the overall correctness theorem still holds if the semantics are corrected because the compiler is always allowed to fail, even on reasonable programs. However, the intermediate results about the front-end become false because the bug is present much earlier in the compiler than the type reconstruction.

Alleviating this bug was essential for proper testing with Csmith, but a full solution requires fixing the semantics, compiler and proofs and is beyond the scope of this work. By implementing an alternative reduction rule in OCaml as a workaround we were able to continue testing without difficulty.

3.3 gcc-torture

The GCC compiler contains an extensive test suite for several of its supported languages [7, §6.4]. The gcc-torture suite contains an executable subset of the tests in C. Many of the test cases use GCC-specific features, but we are able

to reuse a subset selected by another executable C semantics project, `kcc` [6]. Unlike the Csmith generated tests, gcc-torture contains many specialised tests for corner-cases of the C standard. In addition to finding errors, these also serve to highlight deliberate limitations in CompCert's semantics.

A small test harness ran each test case with the executable semantics. The failing cases were manually classified and in a few cases a workaround or fix to the parser was added to prevent known bugs hiding further issues.

The tests revealed that the semantics were missing some minor casting rules and that zero initialisation of file scope variables was not performed when initialising the memory model. Both were handled correctly by the compiler, in the latter case by the informal code that produces the assembler output.

Several issues with the OCaml parser appeared: initialisation of an array by a shorter string did not add null-padding; arrays were not permitted to the left of a `->` dereference; incomplete array types were not fully supported; and the reported line numbers in error messages could be wrong. Ironically, the latter was caused by side-effects and OCaml's own non-deterministic evaluation order.

Deliberate limitations of CompCert were strongly apparent in the result too. Unsupported and partially supported constructs such as `long long` integer types and bitfields caused many test case failures. A more interesting case is that the comparison of pointers is undefined when one points just beyond the end of the object (which is explicitly permitted by the C standard), or when a function pointer is used. However, we were unable to find any mention of this limitation in the documentation, and without testing would never have known about it.

4 Related work

The CompCert developers had already proposed implementing an executable semantics for the first version of Clight, but in a big-step style with a bound on the depth of the evaluation to ensure termination [3, §5.4]. However, it would be difficult to implement the workarounds described in Section 2.4 with a big-step semantics.

Reaction to this work from the CompCert developers has been positive: bugs identified before the next release were fixed (in the case of the missing casts, independently before they were identified by testing), and Leroy has subsequently added a similar small-step interpreter. The new interpreter differs from the present work by focusing solely on the non-deterministic semantics and computing all of the successor states at once. This greatly simplifies the checking and proof of non-stuckness described in Section 2.1, and provides an option to explore the entire space of evaluation orders (which is surprisingly tractable). However, there is no easy way to mimic just the deterministic evaluation order, and no tricks in the driver code to make testing easier. Together with the bug fixes, this makes directly comparing test results with the current work infeasible, although some brief testing with the gcc-torture suite behaved as expected. The latest version of CompCert, 1.11, also features a more efficient memory model, improving the interpreter's performance.

Ševčík et al. [23] have created CompCertTSO, a derivative of an earlier version of CompCert which supports concurrent shared-memory programs with a *Total Store Ordering* (TSO) memory model. The source language for the formalised part of the compiler is ClightTSO and is given an executable semantics in addition to the usual relational semantics, which 'revealed a number of subtle errors.' The main issues found in the present work did not arise because it was based on a version of CompCert that predated their introduction, and the pointer comparisons were not a problem because they were specified differently in CompCertTSO.

Ellison and Roşu [6] have developed a semantics for C based on rewriting logic that aims to be as close to the standard as possible, including many parts of the language that are not currently supported by CompCert. An interpreter called `kcc` is derived from the semantics and has been tested against the gcc-torture test suite that we reused. They go further and perform coverage analysis to gauge the proportion of the semantics exercised by the tests. These semantics are not used for compiler verification, but are intended for studying (and ultimately verifying) the behaviour of C programs.

The Piton compiler [16] was formalised in the Boyer-Moore prover, which is an example of an environment where an executable semantics is the natural choice. The Piton language is a very low-level systems language defined for the project, so they do not benefit from preexisting test suites. However, the work is particularly interesting due to the connected hardware formalisation of the target. A later use of executable semantics in ACL2 demonstrated the usefulness of executable semantics in hardware verification because running an existing test suite for AMD's RTL language against the semantics was crucial for convincing managers that the model was relevant [19, §3].

Lochbihler and Bulwahn [14] applied and extended Isabelle's code extraction and predicate compiler [2] to animate the semantics of a multithreaded Java formalisation in Isabelle/HOL, JinjaThreads. This is a very appealing approach because the predicate compiler deals with most of the burden of writing an executable version of an inductively defined semantics, although JinjaThreads still required a substantial amount of work to deal with some difficult definitions. Their description of testing focuses on performance rather than correctness, but executable versions of the code have been tested on an ongoing basis since the earlier Jinja formalisation that it is based on [9].

5 Conclusions

We have shown that executable semantics can be useful enough for the necessary task of validating semantics to justify retrofitting them to existing verified compilers, and in the case of CompCert found a real compiler bug in the formalised front-end, alongside numerous minor issues. It also illustrates that testing of the semantics can be more sensitive than testing the compiler: our original failing case for the conditional expressions bug would (and did) pass compiler testing,

but using that failure in the semantics we were able to derive a test case that also failed in the compiler.

In addition to the testing, the executable semantics were also useful for demonstrating the limitations of the semantics, both known and unknown. Moreover, we were able to take the opportunity to prove that an intuitive relationship between the deterministic and non-deterministic semantics of CompCert C holds.

References

1. Amadio, R., Asperti, A., Ayache, N., Campbell, B., Mulligan, D., Pollack, R., Régis-Gianas, Y., Coen, C.S., Stark, I.: Certified complexity. Procedia Computer Science 7, 175–177 (2011)
2. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. 5674, pp. 131–146. Springer Berlin / Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03359-9_11
3. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning 43, 263–288 (2009), http://dx.doi.org/10.1007/s10817-009-9148-3
4. Programming languages — C. International standard ISO/IEC 9899:1999, ISO (1999)
5. Campbell, B., Pollack, R.: Executable formal semantics of C. Tech. Rep. EDI-INF-RR-1412, School of Informatics, University of Edinburgh (2010)
6. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 533–544. POPL '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2103656.2103719
7. Free Software Foundation: GNU Compiler Collection (GCC) Internals (2008), version 4.4.3
8. Jourdan, J.H., Pottier, F., Leroy, X.: Validating LR(1) parsers. In: Seidl, H. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 7211, pp. 397–416. Springer Berlin / Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-28869-2_20
9. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst. 28(4), 619–695 (Jul 2006), http://doi.acm.org/10.1145/1146809.1146811
10. Leinenbach, D., Petrova, E.: Pervasive compiler verification from verified programs to verified systems. Electronic Notes in Theoretical Computer Science 217, 23 – 40 (2008), http://www.sciencedirect.com/science/article/pii/S1571066108003836
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52, 107–115 (Jul 2009), http://doi.acm.org/10.1145/1538788.1538814
12. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning 41(1), 1–31 (2008)
13. Letouzey, P.: A new extraction for Coq. In: Types for Proofs and Programs (TYPES 2002). Lecture Notes in Computer Science, vol. 2646, pp. 200–219. Springer-Verlag (2003)

14. Lochbihler, A., Bulwahn, L.: Animating the formalised semantics of a Java-like language. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving, Lecture Notes in Computer Science, vol. 6898, pp. 216–232. Springer Berlin / Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-22863-6_17
15. Milner, R., Weyhrauch, R.: Proving compiler correctness in a mechanized logic. Machine Intelligence 7, 51–70 (1972)
16. Moore, J.S.: A mechanically verified language implementation. Journal of Automated Reasoning 5, 461–492 (1989), http://dx.doi.org/10.1007/BF00243133
17. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) Compiler Construction: 11th International Conference (CC 2002). Lecture Notes in Computer Science, vol. 2304, pp. 213–228. Springer (2002)
18. Rideau, S., Leroy, X.: Validating register allocation and spilling. In: Gupta, R. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 6011, pp. 224–243. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11970-5_13
19. Strother Moore, J.: Symbolic simulation: An ACL2 approach. In: Gopalakrishnan, G., Windley, P. (eds.) Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science, vol. 1522, pp. 530–530. Springer Berlin / Heidelberg (1998), http://dx.doi.org/10.1007/3-540-49519-3_22
20. Tatlock, Z., Lerner, S.: Bringing extensibility to verified compilers. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation. pp. 111–121. PLDI '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1806596.1806611
21. Team, T.C.D.: The Coq Proof Assistant: Reference Manual, Version 8.3. INRIA (2010), http://coq.inria.fr/distrib/8.3pl2/refman/
22. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 17–27. ACM, New York, NY, USA (2008)
23. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 43–54. POPL '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1926385.1926393
24. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. pp. 283–294. PLDI '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1993498.1993532