A Typed C11 Semantics for Interactive Theorem Proving

Robbert Krebbers Freek Wiedijk
ICIS, Radboud University Nijmegen, The Netherlands
mail@robbertkrebbers.nl freek@cs.ru.nl

Abstract

We present a semantics of a significant fragment of the C programming language as described by the C11 standard. It consists of a small step semantics of a core language, which uses a structured memory model to capture subtleties of C11, such as strict-aliasing restrictions related to unions, that have not yet been addressed by others. The semantics of actual C programs is defined by translation into this core language. We have an explicit type system for the core language, and prove type preservation and progress, as well as type correctness of the translation.

Due to unspecified order of evaluation, our operational semantics is non-deterministic. To explore all defined and undefined behaviors, we present an executable semantics that computes a stream of finite sets of reachable states. It is proved sound and complete with respect to the operational semantics.

Both the translation into the core language and the executable semantics are defined as Coq programs. Extraction to OCaml is used to obtain a C interpreter to run and test the semantics on actual C programs. All proofs are fully formalized in Coq.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords ISO C11 Standard, Operational Semantics, Executable Semantics, Interactive Theorem Proving, Coq

1. Introduction

Many programs need a high runtime performance, close control of the underlying hardware, to be able to run with a minimal runtime environment, or to be very portable. All these reasons explain the ongoing popularity of the C programming language, especially for embedded systems and systems programming in general. Of course one pays for these benefits by using a low-level language in which it is very easy to make mistakes with potentially disastrous consequences. This is especially shown by the proliferation of malware on computers all around the world. Without the widespread use of C and its derivative languages, malware would have a much harder time tainting systems everywhere.

An interesting approach to remedy this situation is to use *proofs* to establish the safety of C programs. That way one gets all performance, control and portability benefits of C, without the dangers caused by it. Such approaches range from light weight methods like static analysis (which is by nature incomplete) to systems where a user can interactively reason about programs.

Still with systems of the latter kind, it is often unclear whether the system matches the compiler that is used, or might be used in the future, because most of these systems are about a version of C that is quite specific. Moreover, the semantics of C that such a system embodies is generally not made explicit, which makes it hard to establish that it does not contain semantic errors.

For this reason, the CH_2O project [13–18] aims at developing an *explicit* formal semantics that should match the official description of C, namely the C11 standard [11], as closely as possible. Our goal is that if one proves something about a program with respect to the CH_2O semantics, it will behave that way with *any* C11 compliant compiler. There are two projects that are very close:

- The CompCert project by Leroy *et al.* [20] has a formal semantics for a C-like language called CompCert C in Coq. This language can be compiled with a compiler written in Coq, which has been proved correct with respect to this formal semantics.
- Ellison and Rosu [6] have developed a formal semantics of C in the $\mathbb{K}$-framework. In this work the goal is also to explicitly formally model the C11 standard as closely as possible.

In both projects, as well as in ours, there is a *formal* description of a significant part of C close to the C11 standard and an *executable* interpreter that matches the semantics precisely. However, both projects also differ from our work:

- **Proof infrastructure.** Although Ellison and Rosu support more C features than we do, they do not have proof infrastructure around it. Their semantics, despite being written in a formal framework, should more be seen as a debugger or state space search tool. Our entire semantics, as well as proofs of metatheoretical properties about it, has been formalized in Coq.
- **Explicit typing.** Neither of these projects have an explicit notion of typing, whereas we have defined type judgments. Therefore we have been able to prove properties of our language like type preservation and progress.
- **Formal translation from abstract syntax.** We process the abstract syntax of C to an intermediate language *inside* Coq. We prove that this translation only yields well-typed programs. Although CompCert's parser has been formally verified [12], the majority of the translation from abstract syntax to CompCert C is performed using unverified OCaml code. This translation involves semantic transformations such as evaluation of constant expressions, that we have proved sound and complete.
- **Core language.** Ellison and Rosu do not have an intermediate language: the semantics operates on the abstract syntax of a program itself. We claim that by first going to a *core* language, the description of various semantic features becomes more princi-

CPP '15, January 13–14 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676724.2693571

pled and simpler. For example, this way we do not have duplication in the semantics of various looping constructs.

- **Closer to the C11 standard.** In the CompCert semantics the goal is *not* to make sure that what can be proved according to the semantics will hold for *any* C11 compiler, it just has to hold for one specific compiler: the CompCert compiler. To that end, CompCert is allowed to make choices for implementation defined behavior (*e.g.* integer representations) and gives a semantics to various undefined behaviors (such as aliasing violations).

  Like in our work, Ellison and Rosu's goal is to *exactly* describe the C semantics. However, for some programs their semantics is less precise than ours (see Section 2.4). We can be more precise because in our memory model data objects are structured like trees, while in theirs they consist of flat arrays of bytes.

Approach. To be able to test our semantics, we have developed an 'interpreter'. This is not an ordinary interpreter that executes a program according to *one* interpretation of the C standard, and that has arbitrary behavior in the case of undefined behavior. Instead, our interpreter calculates *all* behaviors of the program that are allowed by the standard, and when the program has undefined behavior, it will explicitly state this undefinedness.

Our interpreter is also different from compilers or interpreters that insert tests for undefined behaviors as a protection. Those compilers or interpreters generally only follow one possible execution order. Instead, our interpreter is not primarily meant to be a debugging tool, but instead is an *exploration* tool, intended to explore the implications of the C standard [5, 30].

We aim to follow the C11 standard [11] as closely as possible. Unfortunately, the standard is often ambiguous due to its use of natural language (*e.g.*, see the message [24] on the standard's committee's mailing list, and Defect Reports #260 and #236 [10]). In the case of ambiguities, we err on the side of caution and favor undefined behavior. Since our interests lie in proving the correctness of C programs with respect to arbitrary C11 compilers, assigning undefined behavior in the case of doubt is the safe choice.

The organization of this paper follows the structure of our C interpreter as shown in Figure 1. We have two languages:

- **CH_2O core C** is described in Section 3. This language is quite abstract, and not very close to actual C.
- **CH_2O abstract C** is described in Section 6. This language is very close to the abstract syntax trees of actual C.

The interpreter then consists of four passes to get from C source code to the behavior of the program.

Related work. Apart from the related work discussed above, there is an abundance of related work on (executable) semantics for large imperative programming languages in proof assistants.

Norrish has defined a semantics of a large part of the C89 type system and semantics in the interactive theorem prover HOL [26]. His main focus was to capture non-determinism in expressions, and our expression semantics is based on his. However, many of the language features we consider are more recent than C89.

Campbell has defined an executable semantics for CompCert C [4] and proved soundness and completeness with respect to the deterministic (leftmost-innermost) operational semantics. The resulting interpreter, that can be used to effectively test the CompCert C semantics, has been reimplemented by Leroy as part of the official CompCert distribution. This reimplementation is also able to handle non-determinism and explore a program's whole state space. Since the amount of non-determinism in CompCert C is more restrained than in our semantics (it is at most finitary branching, whereas we have infinitary branching), our completeness proof with respect to the operational semantics is more involved.

Compared to the CompCert C executable semantics, we handle a larger part of the interpreter inside Coq. In particular, our translation from abstract syntax is implemented in Coq, and the machinery to compute the reachable state set is implemented using a verified implementation of hash-sets in Coq.

There have been many previous efforts on creating sound and complete executable semantics. For example, Zhao *et al.* does this for (a deterministic version of) LLVM [31], Bodin *et al.* for Javascript [2], and Lochbihler and Bulwahn for Java [23]. The latter uses code extraction to automatically obtain an executable semantics from an inductive definition. Due to excessive non-determinism in C, our executable semantics has been written by hand.

There has been a lot of related work on tool support for writing definitions of programming languages which can be automatically translated to different proof assistants. Most notably, there are the OTT [29] and Lem [27] tools. Since we wish to use advanced features of Coq, we have not used such tools.

Contribution. In [14, 18] we have introduced an operational and axiomatic semantics for subtle features of a C-like language. Since the focus of these papers was mostly on separation logic, we used a basic memory model that merely supported integers and pointers. In this paper, we extend the operational semantics as follows:

- We integrate our new memory model that supports array, struct, and union types [13, 15]. As usual, such integration suffers from feature interaction leading to many changes to the operational semantics (like introducing new language constructs), as well as the memory model (like extending the permission system).
- We extend the language with a type system and prove that the operational semantics enjoys type presentations and progress.
- We extend the supported C fragment significantly (*e.g.* typedefs, enums, static and global variables, initializers, constant expressions) by defining a translation from a larger fragment of C. This translation is proved type sound.
- We define an executable semantics that can be used to test the semantics against actual C programs. This executable semantics is proved sound and complete. The completeness proof is non-trivial due to excessive non-determinism in our semantics.
- All definitions and proofs have been formalized using Coq. The development can be extracted to OCaml, resulting in an interpreter that can be used to explore all behaviors of C programs.

This paper describes a large formalization effort, and we only have space to show representative parts of definitions. All details can be found online at:

`http://robbertkrebbers.nl/research/ch2o/`.

2. Challenges

The C11 standard gives compilers considerable freedom in what behaviors a program may have [11, 3.4]. It uses the following notions of under-specification:

- *Unspecified behavior*: two or more behaviors are allowed. For example: the execution order of expressions. The choice may vary for each use of the construct.
- *Implementation defined behavior*: like unspecified behavior, but the compiler has to document its choice. For example: size and endianness of integers.
- *Undefined behavior:* the standard imposes no requirements at all, the program is even allowed to crash. For example: dereferencing a `NULL` pointer or signed integer overflow.

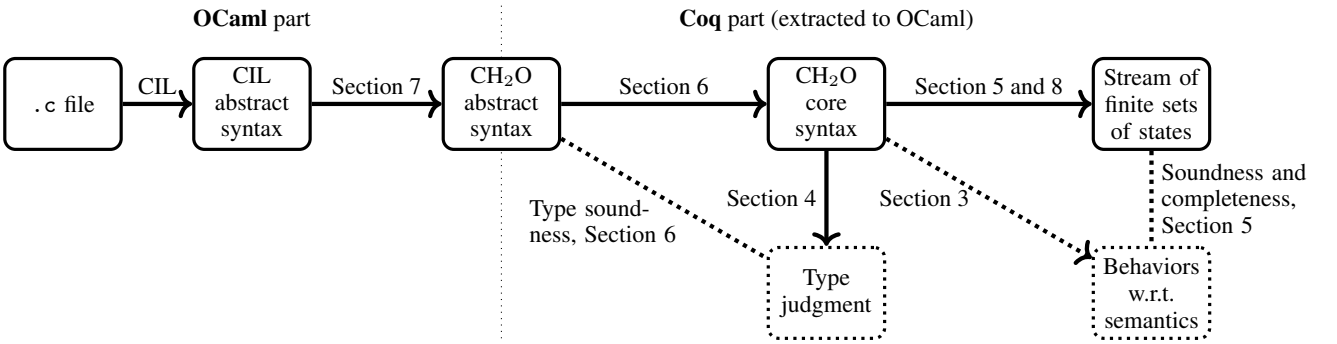


Figure 1. Overview of the architecture of the interpreter.

Under-specification is used extensively to make C portable, and to allow compilers to generate fast code. For example, when dereferencing a pointer, no code has to be generated to check whether the pointer is valid or not. If the pointer is invalid (`NULL` or a dangling pointer), the compiled program may do something arbitrary instead of having to exit with a nice error message.

Since the CH_2O semantics intends to be a formal version of the C11 standard, it has to capture the behavior of *any* C compiler, and has to take *all* under-specification seriously (even if that makes the semantics complex). In this section, we will describe a number of subtle forms of underspecification and give examples of bizarre behaviors exhibited by actual compilers.

2.1 Sequence point violations and non-determinism

Instead of having to follow a specific execution order, in C the execution order of expressions is unspecified. This is a common cause of portability and maintenance problems because a compiler may use an arbitrary execution order for each expression.

To make more effective optimizations possible (*e.g.* delaying of side-effects and interleaving), the C standard requires the programmer to ensure that all execution orders satisfy a certain condition. If this condition is not met, the program has undefined behavior. Let us consider an example where this is the case:

```
int x;
int y = (x = 3) + (x = 4);
printf("%d %d\n", x, y);
```

By considering all possible execution orders, one would naively expect this program to print `4 7` or `3 7`, depending on whether the assignment `x = 3` or `x = 4` is executed first. However, the *sequence point restriction* does not allow an object to be modified more than once (or being read after being modified) between two *sequence points* [11, 6.5p2]. A sequence point occurs for example at the end `;` of a full expression, before a function call, and after the first operand of the conditional `? :` operator [11, Annex C]. Hence, both execution orders lead to a sequence point violation, and therefore result in undefined behavior. Indeed, when compiled with `gcc -O2` (version 4.6.4), the above program prints `4 8`, which does not correspond to any of the execution orders.

Non-determinism in C is even more unrestrained than some may think. The execution order in e_1 `+` e_2 is unspecified, but this does not mean that either e_1 or e_2 will be executed entirely before the other. Instead, it means that execution can be interleaved; first a part of e_1, then a part of e_2, then another part of e_1, and so on. Hence, the following expression is allowed to print `bac`:

```
printf("a") + (printf("b") + printf("c"));
```

Our approach to handling non-determinism and sequence points is inspired by Ellison and Rosu [6] and Norrish [26]. Each object in memory carries a permission that is changed into a locked variant whenever a write occurs. Having a locked permission, the memory model prohibits any access (read or write) to this object. At the next sequence point, the permission is changed back into the unlocked variant, making future accesses possible again.

2.2 Non-local control and block scope variables

C allows unrestricted gotos which (unlike break and continue) may not only jump out of blocks, but can also jump into blocks. Blocks may contain local variables, called *block scope variables*, which can be "taken out of their block" by keeping a pointer to them. Leaving a block results in the memory of these variables being freed, and thus making pointers to them invalid. Consider:

```
int *p = NULL;
l: if (p) { return (*p); }
   else { int j = 10; p = &j; goto l; }
```

When the label `l` is passed for the first time, the variable `p` is `NULL`. Hence, execution continues in the block where `p` is assigned a pointer to the block scope variable `j`. After the `goto l`, the block containing `j` is left, and the memory of `j` is freed. Hence, the conditional `if(p)` on a dangling pointer after the goto yields undefined behavior.

2.3 Indeterminate memory and pointers

A pointer value becomes indeterminate when the object it points to reaches the end of its lifetime [11, 6.2.4] (goes out of scope, or has been deallocated). Not only dereferencing indeterminate pointers has undefined behavior, but also using such pointers in pointer arithmetic and pointer comparisons. For example, consider:

```
int *p = malloc(sizeof(int));
free(p);
int *q = malloc(sizeof(int));
printf("%d\n", p == q);
```

If both calls to `malloc` succeed[1] the program still has undefined behavior because `p` has become indeterminate.

When a pointer has been deallocated, not just the argument of `free` becomes indeterminate, but also all other copies of that pointer. In our memory model we represent pointer values symbolically, and keep track of the memory areas that have been deallocated. The behavior of operations like `==` depends on the memory state, which allows us to accurately capture undefined behaviors.

[1] The size of the CH_2O memory is unbounded, hence `malloc` cannot fail. Addressing finiteness of the memory is left for future work.

2.4 Effective types and aliasing restrictions

Aliasing means that multiple pointers refer to the same object in memory. Consider:

```
int f(int *p, int *q) {
  int x = *q; *p = 10; return x;
}
```

When `f` is called with aliased pointers `p` and `q`, the assignment to `*p` also affects `*q`. As a result, a compiler cannot transform the function body of `f` into the shorter `*p = 10; return (*q);`.

Unlike this example, there are many situations in which pointers can be assumed *not* to alias. It is essential for an optimizing compiler to determine when aliasing cannot occur, and use this information to generate faster code. The technique of determining whether pointers can alias or not is called *alias analysis*.

In *type-based alias analysis*, type information is used to determine if pointers are aliased. Consider the following example:

```
short g(int *p, short *q) {
  short x = *q; *p = 10; return x;
}
```

Here, a compiler should be able to assume that `p` and `q` are not aliased because they point to objects with different types. However, the static type system of C is too weak to ensure that property because a union type can be used to call `g` with aliased pointers.

```
union INT_SHRT { int x; short y; };
union INT_SHRT u = { .y = 3 };
g(&u.x, &u.y);
```

A union is C's version of a *sum* type. Contrary to sum types in many other programming languages, unions are *untagged* instead of *tagged*, which means that the current variant of a union is not stored. As a result, unions destroy the property that each memory area has a unique type that is known at compile time. The *effective type* [11, 6.5p6-7] of a memory area thus depends on the *run time behavior* of the program.

The *strict-aliasing restrictions* [11, 6.5p6-7] imply that a pointer to a variant of a union type (not to the whole union itself) can only be used for an access (read or write) if the union is in that particular variant. Calling `g` with aliased pointers (as in the example where `u` is in the `y` variant, and is accessed through a pointer `p` to the `x` variant) results in undefined behavior.

Contrary to existing formalizations (Leroy *et al.* [21] and Ellison and Rosu [6]) where each object consists of an array of bytes, we use structured trees [13]. By giving the memory more structure, we capture the strict-aliasing restrictions: effective types are modeled by the state of the trees in the memory model.

2.5 Type-punning

Despite the aliasing restrictions of C, it is under certain conditions allowed to access a union through another variant than the current one. This is called *type-punning* [11, 6.5.2.3]. Since the C standard is ambiguous about these conditions[2], we follow the interpretation by the GCC documentation [8]. It states: "type-punning is allowed, provided the memory is accessed through the union type".

For example, according to this interpretation the following program has implementation defined behavior (on architectures where `short`s do not have trap values):

```
union INT_SHRT { int x; short y; };
union INT_SHRT u; u.x = 3;
printf("%d", u.y);
```

Type-punning may only be performed directly via an l-value of union type. Indirect type-punning via a pointer to a particular variant of a union type yields undefined behavior. For example:

```
union INT_SHRT u; short *p = &u.y; u.x = 3;
printf("%d", *p);
```

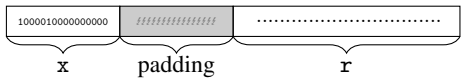
We formalize the informal semantics of GCC by decorating the formal references to subobjects with annotations [13]. Whenever a pointer to a variant of some union is stored in memory, or used as the argument of a function, the annotations are changed to ensure that type-punning is no longer possible via that pointer. In [13], we have shown that the GCC interpretation is correct by proving that a compiler can indeed perform type-based alias analysis [13].

2.6 Byte-level operations

Apart from *high-level* access to objects in memory by means of typed expressions, C also allows *low-level* access by means of byte-wise manipulation. Each object of type τ can be interpreted as an `unsigned char` array of length `sizeof`(τ), which is called the *object representation* [11, 6.2.6.1p4]. An object can thus be copied by copying its object representation. For example:

```
struct { short x, *r; } s1 = { 10, &s1.x }, s2;
for (size_t i = 0; i < sizeof(s1); i++)
  ((unsigned char*)&s2)[i] =
    ((unsigned char*)&s1)[i];
```

The alignment requirements put restrictions on the addresses at which objects may be allocated [11, 6.2.8]. For each type τ, there is an implementation defined integer constant `_Alignof`(τ). Objects of type τ should be allocated at addresses that are a multiple of `_Alignof`(τ). On 32-bits architectures with `_Alignof(short*)`$=4$ (*e.g.* x86), the object representation of `s1` might be:



Due to the alignment requirements, the object representation of the above struct contains a hole. The bytes belonging to these holes are called *padding bytes*, and their values remain indeterminate even after initialization of the whole struct. To facilitate byte-wise copying of structs, a memory model should allow one to read and write all bytes, even those that are indeterminate, using an expression of type `unsigned char` [11, 6.2.6.1p5]. Note that in general, reading indeterminate memory has undefined behavior (for example, reading from an uninitialized `int` variable).

Our memory model uses a symbolic representation of bits in order to distinguish determinate and indeterminate memory. This way, we can precisely keep track of the situations in which reading and writing indeterminate memory is permitted.

The following excerpt from the C11 standard points out another challenge with respect to padding bytes [11, 6.2.6.1p6]:

> When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values

Let us illustrate this difficulty by an example:

```
struct { short x, *r; } s1 = { 10, NULL };
((unsigned char*)&s1)[3] = 1;
s1.x = 11;
printf("%d\n", ((unsigned char*)&s1)[3]);
```

[2] The term *type-punning* is merely used in a footnote, but for the related *common initial segment* rule, it uses the notion of *visible*, which is not clearly defined either.

We assign 1 to the third byte `((unsigned char*)&s1)[3]` of the struct `s1`, and then assign 11 to the field `x` of the struct `s1`. The last assignment makes all padding bytes of `s1` indeterminate, and consequently the argument of `printf` will be indeterminate. In our memory model based on trees, we enforce that padding bytes are always indeterminate, and therefore have this behavior implicitly.

2.7 Arithmetic conversions and overflow

In order to make C portable, the C standard gives compilers a lot of freedom to represent integers and to perform integer arithmetic. In particular, the sizes of integer types are implementation defined. For example, `int` does not necessarily have to be 32 bits and be able to exactly hold values between -2^{31} and $2^{31}-1$. Only some minimum limits are described [11, 5.2.4.2.1]. In order to capture different architectures, our memory model is parametrized by an abstract interface of *integer implementations*.

Overflow of signed integers yields undefined behavior, whereas overflow has defined behavior (and wraps around modulo) for unsigned integers. For example, consider the following function:

```
int f(int x) { return x < x + 1; }
```

A compiler is thus allowed to optimize the function `f` to always return 1. Indeed, when compiled with `gcc -O2` (version 4.6.3), the following program prints `INT_MAX < INT_MAX+1 = 1`.

```
printf("INT_MAX < INT_MAX+1 = %d\n", f(INT_MAX));
```

The *integer promotions* [11, 6.3.1.1p2] and *usual arithmetic conversions* [11, 6.3.1.8p1] make the situations in which signed integer overflow occurs more subtle. For example, the following program has defined behavior (provided that `SHRT_MAX < INT_MAX`) because the arguments of `<` are promoted to `int` type.

```
printf("SHRT_MAX < SHRT_MAX+1 = %d\n",
  SHRT_MAX < SHRT_MAX + 1);
```

2.8 End-of-array pointers

The way the C standard deals with pointer equality is subtle. Consider the following excerpt [11, 6.5.9p6]:

> Two pointers compare equal if and only if [...] or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.

End-of-array pointers are peculiar because they cannot be dereferenced. Nonetheless, their use is common programming practice when looping through arrays.

```
void inc_array(int *p, int n) {
  int *end = p + n;
  while (p < end) (*p++)++;
}
```

End-of-array pointers can also be used in a way that is not stable under compilation. In the example below, the `printf` is executed only if `x` and `y` are allocated adjacently in memory (typically the stack).

```
int x, y;
if (&x + 1 == &y)
  printf("x and y are allocated adjacently\n");
```

Inspired by the CompCert memory model of Leroy *et al.* [21], we represent pointer values symbolically. The use of symbolic pointer representations restricts the situations in which pointer arithmetic and pointer comparisons have defined behavior. This way, we assign undefined behavior to questionable uses of end-of-array pointers while assigning the correct defined behavior to pointer comparisons as in the first example above.

3. Operational semantics of CH_2O core C

In this section we describe the memory model and the operational semantics of our fragment of C11, and show how these address the challenges of Section 2. Since the memory model and operational semantics are an extension of previous work [13–15, 18], we do not describe these in detail. We refer to Figure 2 for the syntax and to the Coq formalization for other details.

3.1 Notations

We let $\mathsf{option}\ B$ denote the *option type over* B, whose elements are inductively defined as either $\bot$ or x for some $x \in B$. Elements of the option type are denoted as $x^? \in \mathsf{option}\ B$. A *partial function* $f : A \to \mathsf{option}\ B$ is called *finite* if its domain is finite. The set of these functions is denoted as $A \to_{\mathsf{fin}} B$.

We let $\mathsf{list}\ B$ denote the *list type over* B, whose elements are inductively defined as either ϵ or $x\,\vec{x}$ for some $x \in B$ and $\vec{x} \in \mathsf{list}\ B$. We let $|\vec{x}| \in \mathbb{N}$ denote the length $\vec{x}$.

3.2 Memory model

Significant existing formal versions of a semantics of C, in particular those by Leroy *et al.* [21] and Ellison and Rosu [6], model the memory as a finite partial function from *object identifiers* to *objects*, where each object consists of a single array of bytes. Every block scope, global and static variable, and invocation of `malloc` is associated a unique object identifier with its own object in memory. This approach separates unrelated objects by construction, and is therefore more suitable for reasoning about memory transformations than the more intuitive approach of modeling the memory as an array of bytes (which is closer to an actual machine).

However, since no information about dynamic types is stored using this approach, not all undefined behaviors related to the aliasing restrictions (see Section 2.4) and interactions between high- and low-level memory accesses (see Section 2.6) can be described accurately. To remedy these shortcomings, we model the memory as a finite partial function from object identifiers to well-typed trees whose structure corresponds to the structure of data types in C. The leafs of these trees consist of arrays of (symbolic) bits that represent base values (integers and pointers).

The key concepts of our memory model are the following.

- *Pointers* contain paths following the structure of types. These paths, called *references*, have annotations $\circ$ and $\bullet$ to restrict the situations in which type-punning has defined behavior (see Section 2.5 for the description of type-punning).
  - The annotation $\bullet$ means that type-punning (accessing another variant of the union than the current one) is allowed.
  - The annotation $\circ$ means that type-punning is forbidden.

  Whenever a pointer is stored in memory, its annotations are set to $\circ$. These annotations give a formal treatment of the strict-aliasing restrictions of C11. Pointers are annotated with the type to which they are cast.
- *Bits* are represented symbolically because each object (including those containing pointers and indeterminate memory) can be interpreted as an `unsigned char` array called the object representation (see Section 2.6). A bit is either a concrete bit 0 or 1, the ith bit $(\mathtt{ptr}\ p)_i$ of a pointer p, or the indeterminate bit $\lightning$. Integers are represented using sequences of concrete bits, and pointers as sequences of abstract pointer bits.

Operators

$$\begin{aligned}
\circledcirc_c \in \mathsf{compop} &::= \texttt{==} \mid \texttt{<=} \mid \texttt{<} \\
\circledcirc_a \in \mathsf{arithop} &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
\circledcirc_b \in \mathsf{bitop} &::= \texttt{\&} \mid \texttt{|} \mid \texttt{\^} \\
\circledcirc_s \in \mathsf{shiftop} &::= \texttt{<<} \mid \texttt{>>} \\
\circledcirc \in \mathsf{binop} &::= \circledcirc_c \mid \circledcirc_b \mid \circledcirc_a \mid \circledcirc_s \\
\circledcirc_u \in \mathsf{unop} &::= \texttt{-} \mid \texttt{\~{}} \mid \texttt{!} \\
\alpha \in \mathsf{assign} &::= \texttt{:=} \mid \circledcirc\texttt{:=} \mid \texttt{:=}\circledcirc
\end{aligned}$$

Types

$$\begin{aligned}
k \in K &:= \text{Set of integer ranks} \\
t \in \mathsf{tag} &:= \text{Set of struct/union names} \\
si \in \mathsf{signedness} &::= \mathsf{signed} \mid \mathsf{unsigned} \\
\tau_\mathsf{i} \in \mathsf{inttype} &::= si\ k \\
\tau_\mathsf{b} \in \mathsf{basetype} &::= \tau_\mathsf{i} \mid \tau* \mid \mathsf{void} \\
\tau, \sigma \in \mathsf{type} &::= \tau_\mathsf{b} \mid \tau[n] \mid \mathsf{struct}\ t \mid \mathsf{union}\ t \\
\Gamma \in \mathsf{env} &:= \mathsf{tag} \to_\mathsf{fin} \mathsf{list\ type}
\end{aligned}$$

Separation algebras and permissions

$$\begin{aligned}
\mathcal{L}(A) &:= \{\circ, \bullet\} \times A && \text{(Lockable SA)} \\
\mathcal{C}(A) &:= \mathbb{Q} \times A && \text{(Counting SA)} \\
\mathcal{T}^x_T(A) &:= A \times T \text{ with } x \in T && \text{(Tagged SA)} \\
p \in \mathsf{perm} &:= \mathcal{L}(\mathcal{C}(\mathbb{Q}))
\end{aligned}$$

Memory model

$$\begin{aligned}
o \in \mathsf{index} &:= \text{Set of memory indexes} \\
r \in \mathsf{refseg} &::= \xhookrightarrow{\tau[n]} i \mid \xhookrightarrow{\mathsf{struct}\ t} i \mid \xhookrightarrow{\mathsf{union}\ t}_q i \text{ with } q \in \{\circ, \bullet\} \\
\vec{r} \in \mathsf{ref} &:= \mathsf{list\ refseg} \\
a \in \mathsf{addr} &::= (o : \tau, \vec{r}, i)_{\sigma > \sigma'} \\
p \in \mathsf{ptr} &::= \mathsf{NULL}\ \tau \mid a \\
b \in \mathsf{bit} &::= 0 \mid 1 \mid (\mathsf{ptr}\ p)_i \mid \lightning \\
\mathbf{b} \in \mathsf{pbit} &:= \mathcal{T}^{\lightning}_{\mathsf{bit}}(\mathsf{perm}) \\
v_\mathsf{b} \in \mathsf{baseval} &::= \mathsf{indet}\,\tau_\mathsf{b} \mid \mathsf{int}_{\tau_\mathsf{i}}\, z \mid \mathsf{ptr}\, p \mid \mathsf{byte}\, \vec{b} \quad \text{with } z \in \mathbb{Z} \\
v \in \mathsf{val} &::= v_\mathsf{b} \mid \mathsf{array}_\tau\ \vec{v} \mid \mathsf{struct}_t\ \vec{v} \\
&\quad \mid \mathsf{union}_t\ (i, v) \mid \overline{\mathsf{union}}_t\ \vec{v} \\
w \in \mathsf{mtree} &::= \mathsf{base}_{\tau_\mathsf{b}}\ \vec{\mathbf{b}} \mid \mathsf{array}_\tau\ \vec{w} \mid \mathsf{struct}_t\ \overrightarrow{w\mathbf{b}} \\
&\quad \mid \mathsf{union}_t\ (i, w, \vec{\mathbf{b}}) \mid \overline{\mathsf{union}}_t\ \vec{\mathbf{b}} \\
m \in \mathsf{mem} &:= \mathsf{index} \to_\mathsf{fin} (\mathsf{mtree} \times \mathsf{bool} + \mathsf{type}) \\
\Omega \in \mathsf{lockset} &:= \mathsf{index} \to_\mathsf{fin} \mathsf{list\ bool}
\end{aligned}$$

CH_2O core C

$$\begin{aligned}
f \in \mathsf{funname} &:= \text{Set of function names} \\
l \in \mathsf{labelname} &:= \text{Set of label names} \\
e \in \mathsf{expr} &::= x_i^\tau \mid [v]_\Omega \mid [a]_\Omega \mid *e \mid \&e \mid e\ \texttt{.}_\mathbf{l}\ r \mid e\ \texttt{.}_\mathbf{r}\ r \\
&\quad \mid [r := e_1]e_2 \mid e_1\ \alpha\ e_2 \mid f(\vec{e}) \mid \mathsf{abort}\ \tau \\
&\quad \mid \mathsf{load}\ e \mid \mathsf{alloc}_\tau\ e \mid \mathsf{free}\ e \mid\ \circledcirc_u\ e \mid e_1 \circledcirc e_2 \\
&\quad \mid e_1\ ?\ e_2 : e_3 \mid (e_1, e_2) \mid (\tau)\, e \\
s \in \mathsf{stmt} &::= e \mid \mathsf{skip} \mid \mathsf{goto}\ l \mid \mathsf{throw}\ n \mid \mathsf{return}\ e \mid l : \\
&\quad \mid \mathsf{local}_\tau\ s \mid \mathsf{catch}\ s \mid s_1\ ;\ s_2 \\
&\quad \mid \mathsf{loop}\ s \mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 \\
\delta \in \mathsf{funenv} &:= \mathsf{funname} \to_\mathsf{fin} \mathsf{stmt}
\end{aligned}$$

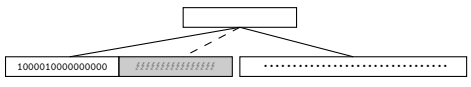
CH_2O core C states

$$\begin{aligned}
\mathcal{E}_\mathsf{s} \in \mathsf{ectx_s} &::= *\square \mid \&\square \mid \square\ \texttt{.}_\mathbf{l}\ r \mid \square\ \texttt{.}_\mathbf{r}\ r \mid [r := \square]e_2 \\
&\quad \mid [r := e_1]\square \mid \square\ \alpha\ e_2 \mid e_1\ \alpha\ \square \\
&\quad \mid f(\vec{e_1}, \square, \vec{e_2}) \mid \mathsf{load}\ \square \mid \mathsf{alloc}_\tau\ \square \mid \mathsf{free}\ \square \\
&\quad \mid\ \circledcirc_u\ \square \mid \square \circledcirc e_2 \mid e_1 \circledcirc \square \\
&\quad \mid \square\ ?\ e_2 : e_3 \mid (\square, e_2) \mid (\tau)\, \square \\
\mathcal{E} \in \mathsf{ectx} &:= \mathsf{list\ ectx_s} \\
\mathcal{S}_\mathsf{s} \in \mathsf{sctx_s} &::= \mathsf{catch}\ \square \mid \square\,;s_2 \mid s_1\,;\square \mid \mathsf{loop}\ \square \\
&\quad \mid \mathsf{if}\ (e)\ \square\ \mathsf{else}\ s_2 \mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ \square \\
\mathcal{S}_\mathsf{e} \in \mathsf{sctx_e} &::= \square \mid \mathsf{return}\ \square \mid \mathsf{if}\ (\square)\ s_1\ \mathsf{else}\ s_2 \\
\mathcal{P}_\mathsf{s} \in \mathsf{ctx_s} &::= \mathcal{S}_\mathsf{s} \mid \mathsf{local}_{o:\tau}\ \square \mid (\mathcal{S}_\mathsf{e}, e) \\
&\quad \mid \mathsf{resume}\ \mathcal{E} \mid \mathsf{params}\ f\ \overrightarrow{o\tau} \\
\mathcal{P} \in \mathsf{ctx} &:= \mathsf{list\ ctx_s} \\
d \in \mathsf{direction} &::= \searrow \mid \nearrow \mid \curvearrowright l \mid \uparrow n \mid \Uparrow v \\
\phi_U \in \mathsf{undef} &::= \lightning\, \mathcal{E}\langle e \rangle \mid \lightning\, \mathcal{S}_\mathsf{e}\langle [v]_\Omega \rangle \\
\phi \in \mathsf{focus} &::= (d,\ s) \mid e \mid \overline{\mathsf{call}}\ f\ \vec{v} \mid \overline{\mathsf{return}}\ f\ v \mid \overline{\mathsf{undef}}\ \phi_U \\
S \in \mathsf{state} &::= \mathbf{S}(\mathcal{P},\ \phi,\ m)
\end{aligned}$$

Figure 2. Syntax of CH_2O core C. In these definitions we let $i, n \in \mathbb{N}$.

By representing bits symbolically, we ensure that additional information that is stored to describe certain undefined behaviors is preserved while performing byte-wise operations on object representations.

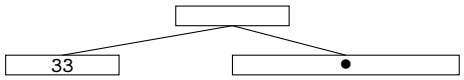
- *Memory trees* are abstract trees whose structure corresponds to the shape of data types. These are used to represent each object in memory. For example, the memory tree corresponding to `struct { short x, *r; } s = { 33; &s.x }` might be:



The leafs of memory trees contain symbolic bits: the integer 33 is represented by its binary representation 1000010000000000, the padding is represented by symbolic indeterminate bits $\lightning$, and the pointer p by a sequence $(\mathsf{ptr}\ p)_0 \ldots (\mathsf{ptr}\ p)_{31}$. These bits are annotated with permissions. Permissions are built modularly using a telescope of separation algebras [15].

The *memory* is a forest of memory trees. Compared to modeling each object in memory as an array of bits, memory trees store the variant of each union, and are explicit about padding bits.

- *Abstract values* are similar to memory trees, but have base values (integers and pointers) on their leafs. The abstract value of `struct { short x, *r; } s = { 33; &s.x }` might be:



Abstract values are used in the external interface of the memory model (which consists of functions to perform stores and loads) and hide internal details of the memory (permissions, padding, and object representations).

The external interface of the CH$_2$O memory model consists of operations with the following types:

$$(\_)\langle\_\rangle_\Gamma : \mathsf{mem} \to \mathsf{addr} \to \mathsf{option\ val} \quad (1)$$

$$\mathsf{force}_\Gamma : \mathsf{addr} \to \mathsf{mem} \to \mathsf{mem} \quad (2)$$

$$(\_)[\_ := \_]_\Gamma : \mathsf{mem} \to \mathsf{addr} \to \mathsf{val} \to \mathsf{mem} \quad (3)$$

$$\mathsf{writable}_\Gamma : \mathsf{addr} \to \mathsf{mem} \to \mathsf{Prop} \quad (4)$$

$$\mathsf{lock}_\Gamma : \mathsf{addr} \to \mathsf{mem} \to \mathsf{mem} \quad (5)$$

$$\mathsf{unlock} : \mathsf{lockset} \to \mathsf{mem} \to \mathsf{mem} \quad (6)$$

$$\mathsf{alloc}_\Gamma : \mathsf{index} \to \mathsf{type} \to \mathsf{bool} \to \mathsf{mem} \to \mathsf{mem} \quad (7)$$

$$\mathsf{freeable} : \mathsf{addr} \to \mathsf{mem} \to \mathsf{Prop} \quad (8)$$

$$\mathsf{free} : \mathsf{index} \to \mathsf{mem} \to \mathsf{mem} \quad (9)$$

The operation $m\langle a\rangle_\Gamma$ yields the value stored at address a in the memory m. This operation yields $\bot$ in case the permissions are not sufficient for a read access, effective types are violated, or in case a is an end-of-array address. Reading from (the abstract) memory is not a pure operation, it may affect the effective types [11, 6.5p6-7]. That means, whenever a union is accessed using an address to a different variant (provided the annotations $\circ$ and $\bullet$ permit it), the variant of the union should be changed in memory. This impurity is factored out by the operation $\mathsf{force}_\Gamma\ a\ m$.

The operation $m[a := v]_\Gamma$ stores the value v at address a in m. A memory store is only permitted in case permissions are sufficient, effective types are not violated, and the address is not end-of-array. These side-conditions are described by $\mathsf{writable}_\Gamma\ a\ m$.

After a successful store, the operation $\mathsf{lock}_\Gamma\ a\ m$ is used to lock the object at address a in m. The lock operation temporarily reduces the permissions of a to prohibit all accesses to a. Locking yields a formal treatment of the sequence point restriction (which states that modifying an object more than once between two sequence points is undefined behavior, see Section 2.1). At a subsequent sequence point (for example, at the end `;` of a full expression, or after the first operand of the conditional `? :` operator has been evaluated), the operation $\mathsf{unlock}\ \Omega\ m$ is used to unlock previously locked objects Ω in memory m. Unlocking has the intended effect of making subsequent accesses possible again.

The operation $\mathsf{alloc}_\Gamma\ o\ \tau\ \beta\ m$ allocates a new object of type τ in the memory m. The new object has object identifier $o \notin \mathsf{dom}\ m$ which is non-deterministically chosen by the operation semantics (see Section 3.3). The boolean β expresses whether the new object is allocated by `malloc` (the expression $\mathsf{alloc}_\tau\ e$ in our language).

Conversely, the operation $\mathsf{free}\ o\ m$ deallocates an object with object identifier o in the memory m. For the case of `malloc`ed memory, the side-condition $\mathsf{freeable}\ a\ m$ ensures that the permissions are sufficient, and that a indeed points to a whole `malloc`ed object with object identifier $\mathsf{index}\ a$.

As part of the Coq development, we have proven many properties about the interaction of memory operations. For example that stores to non-overlapping addresses commute.

3.3 Expression semantics

We define the semantics of expressions and statements by a small step operational semantics. That means, computation is defined as the reflexive transitive closure of a reduction relation. For expressions, we first define head reduction $\rightarrow_\mathsf{h}$, and use *evaluation contexts* (as introduced by Felleisen *et al.* [7]) to select a redex in a whole expression. Given a stack $\rho \in \mathsf{list\ index}$, the rules for head reduction are as follows:

1. $(x_i^\tau, m) \rightarrow_\mathsf{h} ([\mathsf{top}_\tau\ o]_\emptyset, m)$ if $\rho(i) = o$
2. $(*[\mathsf{ptr}\ a]_\Omega, m) \rightarrow_\mathsf{h} ([a]_\Omega, m)$ if a is not a dangling address
3. $(\&[a]_\Omega, m) \rightarrow_\mathsf{h} ([\mathsf{ptr}\ a]_\Omega, m)$
4. $([a]_\Omega\ ._1\ r, m) \rightarrow_\mathsf{h} ([a']_\Omega, m)$
   where a' is obtained by appending r to a
5. $([v]_\Omega\ ._\mathtt{r}\ r, m) \rightarrow_\mathsf{h} ([v']_\Omega, m)$
   where v' is the sub-value at location r in v
6. $([r := [v_1]_{\Omega_1}][v_2]_{\Omega_2}, m) \rightarrow_\mathsf{h} ([v']_{\Omega_1 \cup \Omega_2}, m)$
   where v' is obtained by inserting v_1 at location r in v_2
7. $([a]_{\Omega_1} := [v]_{\Omega_2}, m) \rightarrow_\mathsf{h} ([v]_{\{a\} \cup \Omega_1 \cup \Omega_2}, \mathsf{lock}_\Gamma\ a\ (m[a := v]_\Gamma))$
   provided that $\mathsf{writable}_\Gamma\ a\ m$
8. $(\mathsf{load}\ [a]_\Omega, m) \rightarrow_\mathsf{h} ([v]_\Omega, \mathsf{force}_\Gamma\ a\ m)$ if $m\langle a\rangle_\Gamma = v$
9. $(\mathsf{alloc}_\tau\ [\mathsf{int}_{\tau_i}\ n]_\Omega, m) \rightarrow_\mathsf{h} ([\mathsf{ptr}\ a]_\Omega, \mathsf{alloc}_\Gamma\ o\ (\tau[n])\ \mathsf{true}\ m)$
   for any o satisfying $o \notin \mathsf{dom}\ m$, where a is the 0-th element of $\mathsf{top}_{\tau[n]}\ o$, and provided that $n \neq 0$, and $\mathsf{sizeof}_\Gamma\ \tau \cdot n$ is representable by an integer of type $\mathsf{signed\ ptr\_rank}$
10. $(\mathsf{free}\ [\mathsf{ptr}\ a]_\Omega, m) \rightarrow_\mathsf{h} ([\mathsf{void}]_\Omega, \mathsf{free}\ (\mathsf{index}\ a)\ m)$
    provided that $\mathsf{freeable}\ a\ m$
11. $(\circledcirc_u [v]_\Omega, m) \rightarrow_\mathsf{h} ([\circledcirc_u\ v]_\Omega, m)$
    provided that $\Gamma; m \vdash (\circledcirc_u\ v)\ \mathsf{defined}$
12. $([v_1]_{\Omega_1}\ \circledcirc\ [v_2]_{\Omega_2}, m) \rightarrow_\mathsf{h} ([v_1 \circledcirc v_2]_{\Omega_1 \cup \Omega_2}, m)$
    provided that $\Gamma; m \vdash (v_1 \circledcirc v_2)\ \mathsf{defined}$
13. $(([v]_\Omega, e), m) \rightarrow_\mathsf{h} (e, \mathsf{unlock}\ \Omega\ m)$
14. $([v]_\Omega\ ?\ e_2 : e_3, m) \rightarrow_\mathsf{h} (e_2, \mathsf{unlock}\ \Omega\ m)$ if $\mathsf{istrue}_m\ v$
15. $([v]_\Omega\ ?\ e_2 : e_3, m) \rightarrow_\mathsf{h} (e_3, \mathsf{unlock}\ \Omega\ m)$ if $\mathsf{isfalse}\ v$
16. $((\tau)\ [v]_\Omega, m) \rightarrow_\mathsf{h} ([(\tau)v]_\Omega, m)$
    provided that $\Gamma; m \vdash (\tau)v\ \mathsf{defined}$

(Rule 7 just includes the case of a normal assignment $:=$, the rules for assignment operators $\circledcirc :=$ and $:= \circledcirc$ are similar).

The leafs of expressions, values $[v]_\Omega$ and addresses $[a]_\Omega$, are annotated with a finite set $\Omega \in \mathsf{lockset}$ of *locked* addresses. This set is initially empty, but whenever a write is performed, the written object is locked in memory and its address is added to Ω (see rule 7 for assignment above). Locking a enforces the sequence point restriction because the permission model [14, 15] enforces that consecutive reads and writes to a will fail. Whenever a sequence point occurs, for example in rule 14 for the conditional expression, the locked objects in Ω are unlocked. This makes subsequent accesses to these objects possible again.

The C standard distinguishes *l-values* and *r-values* [11, 6.3.2.1]. This distinction originates from the assignment `e1 = e2` where the left-hand side does not denote an actual value, but an address. L-values (that do not have array type) are implicitly converted to r-values by *l-value conversion* [11, 6.3.2.1p2].

In CH$_2$O core C we make this distinction, as well as the conversion between the two, explicit. L-values reduce to addresses $[a]_\Omega$ while r-values reduce to abstract values $[v]_\Omega$. The expressions $*e$, $\&e$ and $\mathsf{load}\ e$ are used to map between the two. CH$_2$O abstract C has implicit l-value conversion which is being disambiguated during the translation to core C (see Section 6).

CH$_2$O core C uses De Bruijn indices for local variables, which means that a local variable x_i^τ refers to the i-th item on the stack $\rho \in \mathsf{list\ index}$. Accessing the value of a local variable has an extra level of indirection: the stack ρ contains a reference to the value in memory instead of the value of the variable itself. This way, pointers to both local and allocated storage are treated uniformly. Evaluation of a variable x_i^τ consists of looking up its object identifier o in the stack ρ, and returning the address $\mathsf{top}_\tau\ o := (o : \tau, \epsilon, 0)_{\tau > \tau}$ to

the top of the memory tree at position o in memory (rule 1). Using a load, the actual value can be obtained (rule 8).

Due to explicit l-value conversion, CH_2O core C has two operators for field indexing of structs and unions: $e\ ._{\mathsf{l}}\ r$ for l-values, and $e\ ._{\mathsf{r}}\ r$ for r-values. The semantics of these operators are different: the first refines an address to refer to the corresponding field (rule 4), whereas the latter refines the actual value (rule 5).

In the reduction of whole programs (see Section 3.4), we allow $\mathcal{E}[\,e_1\,]$ to reduce to $\mathcal{E}[\,e_2\,]$ provided that $(e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$. To make expression evaluation non-deterministic (see Section 2.1), we include both the contexts $\square$ + e_2 and e_1 + $\square$. However, to enforce that the first operand of the conditional e_1 ? e_2 : e_3 is executed entirely before the others, it is essential that we omit the contexts e_1 ? $\square$: e_3 and e_1 ? e_2 : $\square$. This approach is also used by Norrish [26], Leroy [20] and Ellison and Rosu [6]. Also, like Ellison and Rosu [6], we implicitly use non-determinism to capture undefined behavior due to sequence point violations. For example, in `x + (x = 10)` only one execution order (performing the read after the assignment) leads to a sequence point violation.

The side-condition $\Gamma; m \vdash (v_1 \circledcirc v_2)$ defined for binary operators depends on the memory m (rule 12). This is needed because operations on pointers (for example comparing a pointer, see Section 2.3) only have defined behavior in case the pointers in question have not been deallocated.

The abort expression abort τ does not have a corresponding rule for head reduction and therefore has undefined behavior as explained in Section 3.4. The translation from CH_2O abstract C inserts abort τ expressions in branches of the program that should not be reached to describe all undefined behaviors.

Contrary to our previous work [14], we take the *integer promotions* [11, 6.3.1.1p2] and the *usual arithmetic conversions* [11, 6.3.1.8p1] for operators into account (see Section 2.7). Naively, it seems best to make these promotions and conversions explicit in core C and let the translation of CH_2O abstract C into core C insert appropriate casts. However, for assignment operators it is unclear how this could be handled by translation. For example, consider:

```
int x = INT_MAX;
x += INT_MAX + 2U;
```

This program has defined behavior, and `x` has the value `0` after its execution. Since `INT_MAX + 2U` has type `unsigned int`, the addition `x + (INT_MAX + 2U)` performed by the assignment operator `+=` promotes both arguments to `unsigned int`, and the result will thus wrap around modulo instead of exhibiting undefined behavior. However, if we would have homogeneous assignment operators, then we would have to insert the following `int` cast:

```
int x = INT_MAX;
x += (int)(INT_MAX + 2U);
```

In this case, `(int)(INT_MAX + 2U)` results in signed integer overflow which causes undefined behavior.

3.4 Whole programs semantics

The small step reduction $\rightarrow$ of *program states* $\mathbf{S}(\mathcal{P},\ \phi,\ m)$ uses a *zipper*-like data structure $\mathcal{P}$, called a *program context* [18], which describes the location of the part of the program ϕ, called the *focus*, that is being executed. Program contexts extend the traditional zipper [9] by annotating each block scope variable with its associated object identifier, and furthermore contain the full call stack of the program. We have five kinds of focuses ϕ:

- The focus $(d,\ s)$ describes execution of a *statement* s in *direction* d. The directions correspond to the different forms of (non-local) control: *down* $\searrow$, *up* $\nearrow$, *goto* $\curvearrowright$, *throw* $\uparrow$, or *top* $\Uparrow$.

  When a goto l statement is executed, the direction is changed, and a small step traversal through the zipper is performed until the label l has been reached where normal control flow is resumed. This is described by the following rules:

$$\begin{aligned}
\mathbf{S}(\mathcal{P},\ (\searrow,\ \mathsf{goto}\ l),\ m) &\rightarrow \mathbf{S}(\mathcal{P},\ (\curvearrowright l,\ \mathsf{goto}\ l),\ m)\\
\mathbf{S}(\mathcal{P},\ (\curvearrowright l,\ \mathcal{S}_{\mathsf{s}}[\,s\,]),\ m) &\rightarrow \mathbf{S}(\mathcal{S}_{\mathsf{s}}\,\mathcal{P},\ (\curvearrowright l,\ s),\ m)\\
&\qquad \text{if } l \in \mathsf{labels}\ s\\
\mathbf{S}(\mathcal{S}_{\mathsf{s}}\,\mathcal{P},\ (\curvearrowright l,\ s),\ m) &\rightarrow \mathbf{S}(\mathcal{P},\ (\curvearrowright l,\ \mathcal{S}_{\mathsf{s}}[\,s\,]),\ m)\\
&\qquad \text{if } l \notin \mathsf{labels}\ s\\
\mathbf{S}(\mathcal{P},\ (\curvearrowright l,\ l:),\ m) &\rightarrow \mathbf{S}(\mathcal{P},\ (\nearrow,\ l:),\ m)
\end{aligned}$$

  The point of this traversal is not so much to *search* for the label, but much more to incrementally *calculate* the required allocations and deallocations. Instead, the point of this traversal is to accurately describe the interaction between the goto/break/-continue/return statement and block scope variables (see Section 2.2 for its subtleties).

  When entering a scope with a local variable $\mathsf{local}_\tau\ s$ (this construct is nameless because we use De Bruijn indices for variables), the singular context $\mathsf{local}_{o:\tau}\ \square$ is appended to the head of the program context. The rule for goto is:

$$\begin{aligned}
&\mathbf{S}(\mathcal{P},\ (\curvearrowright l,\ \mathsf{local}_\tau\ s),\ m)\\
&\rightarrow \mathbf{S}((\mathsf{local}_{\tau:o}\ \square)\,\mathcal{P},\ (\curvearrowright l,\ s),\ \mathsf{alloc}_\Gamma\ o\ \tau\ \mathsf{false}\ m)
\end{aligned}$$

  Here, we should have $o \notin \mathsf{dom}\ m$ and $l \in \mathsf{labels}\ s$. After execution of this rule, the program context associates the block scope variable with its object identifier o. The *corresponding stack* $\mathsf{getstack}\,\mathcal{P}$ of a program context $\mathcal{P}$ (which is used by the expression semantics) is obtained by an iteration over $\mathcal{P}$.

- The focus $\overline{\mathsf{undef}}\ \phi_U$ describes undefined behavior. The undefined state ϕ_U indicates how the undefined behavior is caused.
- The focus e describes execution of an expression e. For example, provided $(e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$, we have:

$$\mathbf{S}(\mathcal{P},\ \mathcal{E}[\,e_1\,],\ m_1) \rightarrow \mathbf{S}(\mathcal{P},\ \mathcal{E}[\,e_2\,],\ m_2)$$

  The non-deterministic decomposition in an evaluation context $\mathcal{E}$ and subexpression e_1 models unspecified execution order of expressions.

  In case the expression e_1 is unsafe, *i.e.* it is a redex that cannot be $\rightarrow_{\mathsf{h}}$-reduced (due to undefined behavior), we have:

$$\mathbf{S}(\mathcal{P},\ \mathcal{E}[\,e_1\,],\ m) \rightarrow \mathbf{S}(\mathcal{P},\ \overline{\mathsf{undef}}\ (\lightning\,\mathcal{E}\langle e_1\rangle),\ m)$$

  For function calls we have the following rule:

$$\begin{aligned}
&\mathbf{S}(\mathcal{P},\ \mathcal{E}[\,f([v_0]_{\Omega_0}, \ldots, [v_n]_{\Omega_n})\,],\ m)\\
&\rightarrow \mathbf{S}((\mathsf{resume}\ \mathcal{E})\,\mathcal{P},\ \overline{\mathsf{call}}\ f\ \vec{v},\ \mathsf{unlock}\ (\bigcup \vec{\Omega})\ m)
\end{aligned}$$

  This rule is not part of the expression head reduction $\rightarrow_{\mathsf{h}}$ because it changes the focus to $\overline{\mathsf{call}}\ f\ \vec{v}$.

- The focus $\overline{\mathsf{call}}\ f\ \vec{v}$ describes calling a function f with arguments v. The rule for calling a function is:

$$\mathbf{S}(\mathcal{P},\ \overline{\mathsf{call}}\ f\ \vec{v},\ m_1) \rightarrow \mathbf{S}((\mathsf{params}\ f\ \overrightarrow{o\tau})\,\mathcal{P},\ (\searrow,\ s),\ m_2)$$

  Here, m_2 is obtained by allocating the function arguments v at object identifiers $\vec{o}$ with types $\vec{\tau}$. The reduction relation $\rightarrow$ is parametrized by an environment $\delta \in \mathsf{funname} \rightarrow_{\mathsf{fin}} \mathsf{stmt}$ that maps function names to function bodies. For the above rule, we thus should have $\delta\ f = s$. The program context $\mathsf{params}\ f\ \overrightarrow{o\tau}$ delimits the stack of the callee, and behaves like a sequence $\mathsf{local}_{o_0:\tau_0}\ \square, \ldots, \mathsf{local}_{o_{|\vec{o}|}:\tau_{|\vec{o}|}}\ \square$.

- The focus $\overline{\mathsf{return}}\ f\ v$ describes returning from a function f with return value v. Some of the corresponding rules are:

$$\mathbf{S}((\mathsf{params}\, f\, \overrightarrow{\sigma\tau})\, \mathcal{P}, (\Uparrow v, s), m) \rightarrow \mathbf{S}(\mathcal{P}, \overline{\mathsf{return}}\, v\, , \mathsf{free}\ \vec{o}\ m)$$
$$\mathbf{S}((\mathsf{resume}\ \mathcal{E})\, \mathcal{P}, \overline{\mathsf{return}}\, v\, , m) \rightarrow \mathbf{S}(\mathcal{P}, \mathcal{E}[\,[v]_\emptyset\,], m)$$

Distinguishing different kinds of states for different forms of program execution is inspired by Leroy's CompCert C [19]. However, our treatment of statement execution is entirely different.

The C standard describes while, for, and do-while looping statements [11, 6.8.5]. Adding all of these looping statements as primitives to CH_2O core C would cause duplication (since the semantics of these loops are similar). To that end, we only have a statement $\mathsf{loop}\ s$ for an infinite loop, which combined with the $\mathsf{throw}\ n$ statement that jumps to the nth surrounding $\mathsf{catch}\ s$ block, can encode all C loops (see Section 6 for the translation).

4. Type system of CH_2O core C

Contrary to strongly typed programming languages like Java, ML or Safe Haskell, the C programming language does not enjoy type safety. That means, well-typedness according to the C type system does not ensure the absence of crashes. The reason for this is that even well-typed programs may exhibit executions with undefined behaviors such as dereferences of the NULL pointer, or aliasing or sequence point violations.

Nonetheless, the C type system still ensures the absence of basic mistakes like using an integer where a pointer is expected (without explicit cast) or using an undeclared variable. Clearly, the following program is not valid and should be ruled out by the type system:

```
int f(int x) {
  if(0) { return &x; } else { return 10; }
}
```

We will present a type system for CH_2O core C and prove that it enjoys type preservation and a weak form of type safety. This weak form of type safety guarantees that a well-typed program can keep on reducing, while possibly eventually reaching a final state or an $\overline{\mathsf{undef}}\ \phi_U$ state. From a programmer's point of view, weak type safety gives few guarantees because a program may still crash, but from a formalist point of view, it ensures that our semantics behaves well and that we have not forgotten any reductions.

For each syntax category we introduce a corresponding typing judgment. For example, for types we have the judgment $\Gamma \vdash \tau$ that ensures that τ does not contain any cyclic structs and unions. The typing judgment for expressions is $\Gamma, \Gamma_\mathsf{f}, \Gamma_\mathsf{m}, \vec{\tau} \vdash e : \tau_\mathsf{lr}$, where:

- The map $\Gamma \in \mathsf{tag} \rightarrow_\mathsf{fin} \mathsf{list\ type}$ associates a list of field types to each struct/union name.
- The map $\Gamma_\mathsf{f} \in \mathsf{funname} \rightarrow_\mathsf{fin} (\mathsf{list\ type} \times \mathsf{type})$ associates a list of argument types and a return type to each function name.
- The map $\Gamma_\mathsf{m} \in \mathsf{index} \rightarrow_\mathsf{fin} (\mathsf{type} \times \mathsf{bool})$ associates a type and boolean to each object identifier. The boolean indicates whether the object has been deallocated or is still alive. For technical reasons, we keep track of the types of deallocated objects.
- The list $\vec{\tau} \in \mathsf{list\ type}$ gives the types of the De Bruijn variables.

The type $\tau_\mathbf{l}, \tau_\mathbf{r} \in \mathsf{type} + \mathsf{type}$ indicates whether the expression yields an l-value (address) or r-value (abstract value). We list some typing rules (with typing environments omitted from now on):

$$\frac{\vec{\tau}(i) = \tau}{x_i^{\tau} : \tau_\mathbf{l}} \qquad \frac{e : \tau_\mathbf{l}}{\&e : (\tau*)_\mathbf{r}} \qquad \frac{\Gamma_f(f) = (\vec{\tau}, \sigma) \qquad \vec{e} : \vec{\tau}_\mathbf{r}}{f(\vec{e}) : \sigma_\mathbf{r}}$$

The typing judgment for expression contexts $\mathcal{E} : \tau_\mathsf{lr} \rightarrowtail \sigma_\mathsf{lr}$ states that given an expression e of type τ_lr, the expression $\mathcal{E}[\,e\,]$ obtained by substituting e for the hole $\square$ in $\mathcal{E}$ has type σ_lr.

The typing judgment for statements is of the shape $s : (\beta, \tau^?)$. Here, $\tau^? \in \mathsf{option\ type}$ denotes the type of the return statements in s, and is $\bot$ if s does not contain any returns, and $\beta \in \mathsf{bool}$ denotes whether s is statically known to never reach the end because it always executes a return statement. We list some typing rules:

$$\frac{}{\mathsf{skip} : (\mathsf{false}, \bot)} \qquad \frac{e : \tau_\mathbf{r}}{\mathsf{return}\ e : (\mathsf{true}, \tau)} \qquad \frac{}{\mathsf{goto}\ l : (\mathsf{true}, \bot)}$$

$$\frac{s_1 : (\beta_1, \tau_1{}^?) \qquad s_2 : (\beta_2, \tau_2{}^?) \qquad \tau_1^? \cup \tau_2^? = \sigma^?}{s_1\,;\ s_2 : (\beta_2, \sigma^?)}$$

Here, $\cup : \mathsf{option\ type} \rightarrow \mathsf{option\ type} \rightarrow \mathsf{option}\ (\mathsf{option\ type})$ is defined as $\tau \cup \tau := \tau$, $\bot \cup \tau_\mathsf{lr} := \tau_\mathsf{lr}$, and $\tau_\mathsf{lr} \cup \bot := \tau_\mathsf{lr}$.

The judgment $\delta : \Gamma_\mathsf{f}$ for function environments ensures that each function body in $\delta \in \mathsf{funname} \rightarrow_\mathsf{fin} \mathsf{stmt}$ is well-typed with respect to its prototype in $\Gamma_\mathsf{f} \in \mathsf{funname} \rightarrow_\mathsf{fin} (\mathsf{list\ type} \times \mathsf{type})$. Finally, the judgment $S : g$ ensures that all components of the state S are typed with respect to a main function $g \in \mathsf{funname}$ (in CH_2O core C, `main` can be any function g of any type, so we need to ensure that the argument and return types match).

Lemma 4.1. *All typing judgments satisfy uniqueness of typing and have a corresponding type inference function.*

Proof. Since all type judgments are defined in a syntax directed fashion, this follows trivially. □

Fact 4.2. *All typing judgments are closed under weakening.*

Lemma 4.3 (Type preservation)**.** *If a state S_1 with $S_1 : g$ and $S_1 \rightarrow S_2$, then for resulting state S_2 we have $S_2 : g$.*

Proof. By case analysis on the derivation of $S_1 \rightarrow S_2$, using the fact that all memory operations preserve typing. □

Lemma 4.4 (Weak progress)**.** *If a state is typed $S_1 : g$, then either:*

1. *It can reduce further, that is $S_1 \rightarrow S_2$ for some S_2.*
2. *It is an undefined state, that is $S_1 = \mathbf{S}(\mathcal{P}, \overline{\mathsf{undef}}\ \phi_U, m)$ for some $\mathcal{P}$, ϕ_U and m.*
3. *It is a final state, that is $S_1 = \mathbf{S}(\epsilon, \overline{\mathsf{return}}\, g\, v, m)$ for some v and m.*
4. *The labels are incorrect, that is $S_1 = \mathbf{S}(\mathcal{P}, (\curvearrowright l, s), m)$ for some $\mathcal{P}, l, s$ and m with $l \notin \mathsf{labels}\ s \cup \mathsf{labels}\ \mathcal{P}$.*
5. *The throws are incorrect, that is $S_1 = \mathbf{S}(\mathcal{P}, (\uparrow n, s), m)$ for some $\mathcal{P}, n, s$ and m with $n \not< \mathsf{num\_catches}\ \mathcal{P}$.*

Proof. By case analysis on the structure of S_1. □

For technical reasons, we do not let the judgment $S_1 : g$ ensure correctness of gotos and throws. To that end, the progress theorem includes two cases to account for stuck non-local control. However, the judgment $\delta : \Gamma_\mathsf{f}$ for function environments ensures that all gotos and throws are correct. To that end, whenever we start from an initial state, these cases of stuck non-local control cannot happen.

Theorem 4.5 (Weak type safety)**.** *Given an initial state S_1 for main g with arguments $\vec{v}$, then if $S_1 \rightarrow^* S_2$ we have either:*

1. *$S_2 \rightarrow S_3$ for some S_3.*
2. *$S_2 = \mathbf{S}(\mathcal{P}, \overline{\mathsf{undef}}\ \phi_U, m)$ for some $\mathcal{P}, \phi_U$ and m.*
3. *$S_2 = \mathbf{S}(\epsilon, \overline{\mathsf{return}}\, g\, v, m)$ for some v and m.*

Proof. Using Lemmas 4.4 and 4.3, and the fact that $\rightarrow$ preserves validity of gotos and throws. □

5. Executable semantics of CH_2O core C

The semantics of CH_2O core C (Section 3) is defined as an inductively defined reduction relation $\_ \rightarrow \_ : \mathsf{state} \rightarrow \mathsf{state} \rightarrow \mathsf{Prop}$. Since this relation is not executable, we also define a corresponding function $\mathsf{exec} : \mathsf{state} \rightarrow \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$ that *computes* the finite set of consecutive states.

In previous work [13], we already defined basic operations, such as accessing the memory or performing a cast, as Coq functions that are effectively executable. Hence, creating an executable version of our semantics seems straightforward. However, non-determinism makes the situation more complicated.

- Execution of expressions is non-deterministic. For example, in `(*p = 1) + (*q = 2)`, the assignments may be executed in any order. This occurs in the following rule:

$$\mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_1\,],\, m_1) \rightarrow \mathbf{S}(\mathcal{P},\, \mathcal{E}[\,e_2\,],\, m_2)$$

  Here, an expression is decomposed non-deterministically in an evaluation context $\mathcal{E}$ and a subexpression e_1.

- The choice of picking an object identifier for newly allocated memory is non-deterministic. For example, this occurs in the rule for entering a block scope with a local variable:

$$\begin{aligned}&\mathbf{S}(\mathcal{P},\, (\searrow,\, \mathsf{local}_\tau\, s),\, m)\\ \rightarrow\ &\mathbf{S}((\mathsf{local}_{o:\tau}\, \square)\, \mathcal{P},\, (\searrow,\, s),\, \mathsf{alloc}_\Gamma\, o\, \tau\, \mathsf{false}\, m)\end{aligned}$$

  Here, any unused object identifier $o \notin \mathsf{dom}\ m$ may be chosen.

The first source of non-determinism is finitary because expressions can only be decomposed in a finite number of ways. However, there is an infinite choice of picking fresh object identifiers.

In order to deal with the first source of non-determinism, we define a function $\mathsf{redexes} : \mathsf{expr} \rightarrow \mathcal{P}_{\mathsf{fin}}(\mathsf{ectx} \times \mathsf{expr})$ that decomposes an expression into a finite set of all possible combinations of evaluation contexts and redexes.

Lemma 5.1 (Soundness and completeness of redex splitting)**.** *We have $(\mathcal{E},\, e) \in \mathsf{redexes}\ e'$ iff $e' = \mathcal{E}[\,e\,]$ and e is a redex.*

The second source of non-determinism is more difficult to deal with. Since there is an infinite choice of fresh object identifiers, we cannot just try them all. Instead, we will just pick one:

$$\begin{aligned}&\mathsf{exec}\ (\mathbf{S}(\mathcal{P},\, (\searrow,\, \mathsf{local}_\tau\, s),\, m)) :=\\ &\quad \mathbf{let}\ o := \mathsf{fresh}\ m\ \mathbf{in}\\ &\quad \{\mathbf{S}((\mathsf{local}_{\tau:o}\, \square)\, \mathcal{P},\, (\searrow,\, s),\, \mathsf{alloc}_\Gamma\, o\, \tau\, \mathsf{false}\, m)\}\end{aligned}$$

Using a canonical object identifier for newly allocated memory removes the second source of non-determinism entirely. For the soundness theorem, this choice does not matter.

Theorem 5.2 (Soundness)**.** *If $S_2 \in \mathsf{exec}\ S_1$, then $S_1 \rightarrow S_2$.*

Proof. By case analysis on S_1 using Lemma 5.1. □

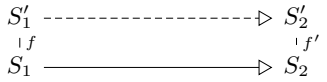
The converse of the above theorem is not true. Given a reduction $S_1 \rightarrow S_2$, we do not necessarily have $S_2 \in \mathsf{exec}\ S_1$ because the operational semantics may have used a different object identifier. For completeness, we show that this choice does not matter.

Definition 5.3. *A state S_1 is an f-*permutation *of state S_2, notation $S_1 \sim_f S_2$, if S_2 is obtained by renaming the object identifiers in S_1 with respect to $f : \mathsf{index} \rightarrow \mathsf{option}\ \mathsf{index}$.*

Fact 5.4. *We have the following properties:*

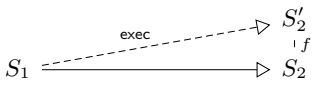
1. Identity: $S \sim_f S$ *for* $f\ x = x$ *for each* $x \in \mathsf{dom}\ S$.
2. Composition: *If* $S_1 \sim_f S_2$ *and* $S_2 \sim_{f'} S_3$, *then* $S_1 \sim_{f' \circ f} S_3$.
3. Symmetry: *If* $S_1 \sim_f S_2$, *then* $S_2 \sim_{f^{-1}} S_1$.
4. Weakening: *If* $S_1 \sim_f S_2$ *and* $f' \supseteq f$, *then* $S_1 \sim_{f'} S_2$.

Lemma 5.5. *If $S_1 \rightarrow S_2$ and $S_1' \sim_f S_1$, then there exists an $f' \supseteq f$ and S_2' such that:*



Proof. By case analysis on the derivation of $S_1 \rightarrow S_2$. □

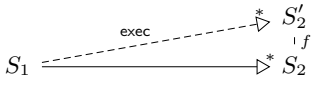
Lemma 5.6. *If $S_1 \rightarrow S_2$, then there exists an f and S_2' such that:*



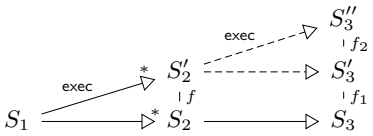
Here, $S_1 \overset{\mathsf{exec}}{\rightarrow} S_2'$ denotes $S_2' \in \mathsf{exec}\ S_1$.

Proof. By case analysis on the derivation of $S_1 \rightarrow S_2$. Lemma 5.1 is used for expression steps, and properties of the memory model for steps that involve allocation of new objects. □

Theorem 5.7 (Completeness)**.** *If $S_1 \rightarrow^* S_2$, then there exists an f and S_2' such that:*



Proof. By induction from the right on the derivation of $S_1 \rightarrow^* S_2$. In the inductive case, we have $S_1 \rightarrow^* S_2 \rightarrow S_3$. By the induction hypothesis we obtain an f and S_2' with $S_1 \overset{\mathsf{exec}}{\rightarrow}{}^* S_2'$ and $S_2' \sim_f S_2$. Next, we obtain an f_1 and S_3' with $S_2' \rightarrow S_3'$ and $S_3' \sim_{f_1} S_3$ by Lemma 5.5. Using Lemma 5.6, we obtain an f_2 and S_3'' with $S_2' \overset{\mathsf{exec}}{\rightarrow} S_3''$ and $S_3'' \sim_{f_2} S_3'$. Displayed as a diagram:



By composition of permutations, we have $S_3'' \sim_{f_1 \circ f_2} S_3$, which concludes the proof. □

6. CH_2O abstract C

CH_2O abstract C bridges the gap between the abstract syntax tree obtained from the parser (Section 7) and CH_2O core C. Its syntax therefore closely resembles the structure of a `.c` file. In particular, it uses named variables instead of De Bruijn indices, makes type annotations (such as those on variables) implicit, and extends CH_2O core C with various features (such as enums and C-like loops). Its semantics is specified by translation into core C. In this section we will highlight some differences between both languages and prove type soundness of the translation.

Like a `.c` file, the syntax of CH_2O abstract C (see Figure 3) consists of a sequence of declarations:

- The items $(s,\, \mathsf{struct}\ \overrightarrow{\tau\, x})$ and $(u,\, \mathsf{union}\ \overrightarrow{\tau\, x})$ declare a struct s or union u with fields $\vec{x}$ of type $\vec{\tau}$.
- The item $(t,\, \mathsf{typedef}\ \tau)$ declares a *typedef* t abbreviating τ.
- The item $(u,\, \mathsf{enum}\ \overrightarrow{x := e^?} : \tau_\mathsf{i})$ declares an *enumeration type* u. This declares constants $\vec{x}$ of integer type τ_i whose values are specified by the constant expressions $\vec{e^?}$. The type τ_i is not fixed because it may differ for each enum [11, 6.7.2.2p4].

$$
\begin{array}{rcl}
x \in \mathsf{string} & ::= & \text{Set of strings} \\
k \in \mathsf{cintrank} & ::= & \mathsf{char} \mid \mathsf{short} \mid \mathsf{int} \mid \mathsf{long} \mid \mathsf{long\ long} \mid \mathsf{ptr} \\
si \in \mathsf{signedness} & ::= & \mathsf{signed} \mid \mathsf{unsigned} \\
\tau_{\mathsf{i}} \in \mathsf{cinttype} & ::= & si^{?}\ k \\
\tau \in \mathsf{ctype} & ::= & \mathsf{void} \mid \mathsf{def}\ x \mid \tau_{\mathsf{i}} \mid \tau* \mid \tau[e] \\
& & \mid \mathsf{struct}\ x \mid \mathsf{union}\ x \mid \mathsf{enum}\ x \mid \mathsf{typeof}\ e \\
e \in \mathsf{cexpr} & ::= & x \mid \mathsf{const}_{\tau_{\mathsf{i}}}\ z \mid \mathsf{sizeof}\ \tau \mid \tau_{\mathsf{i}}\ \mathsf{min} \\
& & \mid \tau_{\mathsf{i}}\ \mathsf{max} \mid \tau_{\mathsf{i}}\ \mathsf{bits} \mid \&e \mid *e \mid e_1\ \alpha\ e_2 \\
& & \mid x(\vec{e}) \mid \mathsf{abort} \mid \mathsf{alloc}_{\tau}\ e \mid \mathsf{free}\ e \\
& & \mid\ \circledcirc_u\ e \mid e_1 \circledcirc e_2 \mid e_1\ \&\&\ e_2 \mid e_1\ \mid\mid\ e_2 \\
& & \mid e_1\ ?\ e_2 : e_3 \mid (e_1, e_2) \mid (\tau)\ I \mid e\ .\ x \\
r \in \mathsf{crefseg} & ::= & [e] \mid .x
\end{array}
$$

$$
\begin{array}{rcl}
I \in \mathsf{cinit} & ::= & e \mid \{\overrightarrow{r := I}\} \\
sto \in \mathsf{cstorage} & ::= & \mathsf{static} \mid \mathsf{extern} \mid \mathsf{auto} \\
s \in \mathsf{cstmt} & ::= & e \mid \mathsf{skip} \mid \mathsf{goto}\ x \mid \mathsf{break} \mid \mathsf{continue} \\
& & \mid \mathsf{return}\ e^{?} \mid \{s\} \mid \overrightarrow{sto}\ \tau\ x := I^{?}\ ;\ s \\
& & \mid \mathsf{typedef}\ x := \tau\ ;\ s \mid s_1\ ;\ s_2 \mid x : s \\
& & \mid \mathsf{while}(e)\ s \mid \mathsf{for}(e_1\ ;\ e_2\ ;\ e_3)\ s \\
& & \mid \mathsf{do}\ s\ \mathsf{while}(e) \mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 \\
d \in \mathsf{decl} & ::= & \mathsf{struct}\ \overrightarrow{\tau\ x} \mid \mathsf{union}\ \overrightarrow{\tau\ x} \mid \mathsf{typedef}\ \tau \\
& & \mid \mathsf{enum}\ \overrightarrow{x := e^{?}} : \tau_{\mathsf{i}} \mid \mathsf{global}\ I^{?} : \overrightarrow{sto}\ \tau \\
& & \mid \mathsf{fun}\ (\overrightarrow{\tau\ x^{?}})\ s^{?} : \overrightarrow{sto}\ \tau \\
\Theta \in \mathsf{decls} & ::= & \mathsf{list}\ (\mathsf{string} \times \mathsf{decl})
\end{array}
$$

Figure 3. Syntax of CH_2O abstract C. The operators are shared with core C, see Figure 2.

- The item $(x, \mathsf{global}\ I^{?} : \overrightarrow{sto}\ \tau)$ declares a global variable x with initializer $I^{?}$.
- The item $(f, \mathsf{fun}\ (\overrightarrow{\tau\ x^{?}})\ s^{?} : \overrightarrow{sto}\ \tau)$ declares a function f with return type τ and arguments $\vec{x}$ of type $\vec{\tau}$. If the function body $s^{?}$ is omitted, the function can already be used in consecutive (mutually recursive) function declarations but has to be redeclared with a corresponding function body later.

The translator from CH_2O abstract C into core C turns a sequence of declarations Θ into:

- An environment $\Gamma \in \mathsf{env}$ of struct/union declarations.
- An environment $\Gamma_{\mathsf{f}} \in \mathsf{funenv}$ of function types.
- An initial memory $m \in \mathsf{mem}$ that contains storage for global and static variables declared in Θ.
- A function environment $\delta \in \mathsf{funname} \to_{\mathsf{fin}} \mathsf{stmt}$ that contains statements for all function declarations in Θ.

An important part of the translation into CH_2O core is evaluation of constant expressions [11, 6.6]. For example, an array type $\tau[e]$ may use an arbitrary expression e to specify the length. The type `int[sizeof(x) + 10]` is thus valid C.

To evaluate such expressions during translation, we have defined an evaluator $[\![\_]\!]_{\Gamma,\rho,m} : \mathsf{expr} \to \mathsf{option}\ (\mathsf{addr} + \mathsf{val})$ for constant expressions. It returns an address or value depending on whether the expression is an l- or r-value. An improvement on work by others is that we have proved soundness and completeness of this evaluator with respect to the operational semantics.

Lemma 6.1. *Constant expression evaluation is sound and complete. That is, given a constant expression e, then:*

$$[\![e]\!]_{\Gamma,\mathsf{getstack}\,\mathcal{P},m} = v \quad \text{iff} \quad \mathbf{S}(\mathcal{P}, e, m) \twoheadrightarrow^{*} \mathbf{S}(\mathcal{P}, v, m).$$

The translator transforms traditional C looping constructs into primitive constructs. For example, $\mathsf{while}(e)\ s$ is translated into:

$$\mathsf{catch}\ (\mathsf{loop}\ (\mathsf{if}\ (e')\ \mathsf{skip}\ \mathsf{else}\ \mathsf{throw}\ 0\ ;\ \mathsf{catch}\ s'))$$

Here, e' and s' are the translations of e and s, respectively. The break and continue statements in s are translated into throw 1 and throw 0, that jump to the outer, respectively the inner catch.

As noted in Section 3, C makes a distinction between l-values (that denote addresses) and r-values (that denote abstract values). Since this distinction is implicit in CH_2O abstract C, but explicit in core C, we have to disambiguate it. In particular, we have to insert casts to perform *l-value conversion* [11, 6.3.2.1p2]. For example, the expression $*p := y$ is translated into:

$$*(\mathsf{load}\ x_i^{\tau *}) := \mathsf{load}\ x_j^{\tau}$$

Here, i and j are the De Bruijn indices corresponding to p and y. Also, we disambiguate field indexing $e\ .\ i$ into $e'\ ._{\mathsf{l}}\ r$ and $e'\ ._{\mathsf{r}}\ r$ depending on whether e is an l- or r-value.

In abstract C, the integer literal `0` is overloaded to denote both the NULL pointer and the integer constant 0 [11, 6.3.2.3p3]. We have to disambiguate this form of overloading in the translation to core C, because the typing rules of various expressions have special cases for NULL pointers [11, 6.5].

A subtle part is the translation of compound literals. Consider:

```
struct S { int x, y, z; } s = { .y=10, .x=11 };
```

Here, fields can be initialized in any order, and fields that are not initialized receive the value 0. We translate such initializers into sequence of $[\_ := \_]\_$ inserts into val_0 (struct S), the struct value initialized with zeros.

Finally, C provides many macros like `INT_MIN` to obtain information about the sizes of integer types [11, 5.2.4.2.1]. CH_2O abstract C has primitive constructs for these macros, such as $\tau_{\mathsf{i}}\ \mathsf{min}$. Since our translator is parametrized by an environment that describes the integer sizes of the architecture, these constructs will be translated into a value that depends on the architecture.

Theorem 6.2 (Type soundness)**.** *If the translator succeeds with environments Γ, Γ_{f}, δ, and initial memory m, they are well-typed.*

7. Testing the semantics

In order to compute the behaviors of a C program, we have turned the single step function $\mathsf{exec} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$ into two functions to compute a stream of reachable states.

- The function $\mathsf{all} : \mathcal{P}_{\mathsf{fin}}(\mathsf{state}) \to (\mathcal{P}_{\mathsf{fin}}(\mathsf{state}) \times \mathcal{P}_{\mathsf{fin}}(\mathsf{state}))^{\omega}$ computes a stream of all reachable states. The nth element $(\mathbf{S}, \mathbf{N})$ of the stream $(\mathsf{all}\ \mathbf{I})$ contains the intermediate states $\mathbf{S} \subseteq \mathsf{state}$ and normal forms $\mathbf{N} \subseteq \mathsf{state}$ after n steps starting in initial states $\mathbf{I} \subseteq \mathsf{state}$.
- The function $\mathsf{some} : (\mathbb{N} \to \mathbb{N}) \to \mathsf{state} \to (\mathsf{state} + \mathsf{state})^{\omega}$ computes a specific execution stream. Given a selection function $f : \mathbb{N} \to \mathbb{N}$ that specifies which redexes should be chosen, the stream $(\mathsf{some}\ f\ I)$ represents a trace starting in $I \in \mathsf{state}$.

In the disjoint union ($\mathsf{state} + \mathsf{state}$), the left variant is an intermediate state, while the right variant is a normal form. Once in normal form, the stream stays constant.

We have extracted the above Coq functions to OCaml and combined this code with the CIL parser by Necula *et al.* [25], using a small layer of OCaml glue[3]. That way we can test our semantics on actual source files. Since the syntax of CH_2O abstract C is close to the structure of `.c` files, the translation from CIL abstract syntax trees to CH_2O abstract syntax is straightforward.

The extracted Coq code, combined with the OCaml layer which loops through the computed streams, yields an interpreter called `ch2o`. This interpreter has three modes: first execution order, random execution order (`-r`), and all possible execution orders (`-t`). For example, given the program:

```
int f(int *p, int z) { return *p = z; }
int main() {
  int x, y = f(&x, 3) + f(&x, 4) + f(&x, 3);
  return x;
}
```

The output of this program using `ch2o -t` is:

```
......+;!|???***%%%%%%****??|||?????|||||!;::::;
!!;::++++--,,,,,,,,,,,,,,
"" 3
"" 4
```

In the output, the density of the symbols indicate the size of the reachable state sets, *e.g.* "`,`" means 2 states, and "`%`" means between 1025 and 4096 states. The strings `""3` and `""4` indicate the possible behaviors of the program: it either has the empty output and returns `3`, or it has the empty output and returns `4`. Although `x` is modified multiple times in the same expression, this program has defined behavior because function calls have a sequence point.

We filter out duplicate states for efficiency (using hash maps implemented and verified in Coq, see Section 8). Therefore, the number of reachable states reduces near the end, and the behavior `""3` is not included twice.

To test the semantics against actual C programs, we need the `printf` function that prints a string to the standard output and returns the length of the printed string [11, 7.21.6.3]. Since strings and I/O are not yet part of the CH_2O semantics, we have temporarily implemented these using a trick in the OCaml layer. The OCaml layer translates each `printf` call into a call to a function that computes the length of the resulting string without actual side-effects. When a `printf` calls occurs during execution, the OCaml loop handles the actual print. We made a small modification to the interpreter, as we store the output state with the CH_2O states, but this has not affected the CH_2O core semantics.

We have tested our semantics using a small test suite for both defined and undefined behavior. This test suite is partially based on subtle examples from the C standard [11], C defect reports [10], and tests by Ellison and Rosu [6]. Since we formally proved many properties of our language [13–15, 18], there were not many bugs. Bugs that we discovered were not fundamental, and related to forgotten implicit casts, or too lenient restrictions on integer operations.

The CH_2O operational semantics is defined with clarity and its use for interactive theorem proving in mind, rather than taking computational efficiency seriously. Although our executable semantics is a naive implementation of it that just performs single steps, it is still efficient enough explore the state space of small programs.

[3] In fact, we use a fork of CIL by CompCert that does not perform any simplifications of the C abstract syntax trees.

The CH_2O interpreter is significantly faster than the semantics of Ellison and Rosu. For example, calculating all behaviors of the following seemingly innocent but very non-deterministic program:

```
int x = 10, *p = &x;
return *p + *p + *p + *p + *p + *p;
```

takes 8s using our semantics and 16m using the Ellison and Rosu semantics. This difference is explained by the fact that we avoid the overhead of a framework like the $\mathbb{K}$-framework by using Coq's extraction mechanism and the optimizing OCaml compiler.

Due to the complex nature of the CH_2O memory model, our interpreter is slower than CompCert's on programs with large arrays or structs (that we represent as lists). Calculating one execution stream of an obfuscated C code contest submission to compute 590 decimals of π by the second author[4], takes 20m using the CH_2O interpreter, 1m using the CompCert interpreter, and did not terminate within reasonable time using the Ellison and Rosu semantics.

8. Implementation and formalization in Coq

All proofs in this paper have been fully formalized using the Coq proof assistant. We used the extraction mechanism of Coq [22] to obtain OCaml code corresponding to the translator from CH_2O abstract C and the executable semantics.

The Coq development involves a significant amount of monadic programming. Multiple components use the *option* monad for partial functions, the executable semantics uses the *set* monad for the set of reachable states, and the translator uses the *exception* monad to propagate error messages. Coq's type class mechanism is used to overload monadic notations such as the *do* notation.

Type classes are also used to let the Coq development correspond well to the definitions on paper and to parametrize the whole development by an abstract interface that describes implementation defined properties (such as the sizes of integers).

The Coq development makes heavy use of finite sets and finite maps. These data structures are implemented as radix-2 search trees in canonical form to obtain both logarithmic time operations and extensional equality. To make reasoning about finite maps and sets convenient, we have developed a substantial library of theory and automation to abstract from implementation details. Radix-2 search trees are also used to implement hash-maps, which are used to filter out duplicates in the reachable state set of the interpreter.

Because the semantics described in this paper is very large, it is quite cumbersome to prove properties about it without automation. To this end, we have combined interactive proofs with automated proofs using `Ltac`.

The proof of completeness (Theorem 5.7) uses in the Coq development a more general notion of *memory refinements* of which the permutations of Definition 5.3 are an instance. Using this more general notion we have not just established that the semantics is invariant under renaming, but also under other internal properties of the memory representation that should not be exposed.

The Coq development, which is entirely constructive and axiom free, consists of the following parts:

Component	**LOC**
Prelude (lists, monads, maps, sets, hash tables, ...)	12.218
Memory (pointers, values, memory trees, ...)	13.686
Operational semantics (type safety, ...)	4.822
Refinements	5.631
Executable semantics (soundness & completeness)	456
Translator from abstract C into core C	1.778
OCaml glue (translator from CIL, `printf` hack, ...)	1.014
Total	39.605

[4] A C11 version of `http://www.cs.ru.nl/~freek/notes/pi.c`.

9. Conclusion

We have defined an operational and executable semantics for a significant fragment of C11, that have been proved sound and complete with respect to each other in Coq. Contrary to previous work, we have used a typed core language, and specified the semantics of C programs by translation into this core language. This translation is implemented as a Coq program that has been proved sound.

Using the extraction mechanism of Coq this yields an interpreter that can explore the full state-space of a C program. We followed the C11 standard closely, and as a result our interpreter can detect subtle undefined behaviors not yet addressed by others.

Since C is a large and subtle language, there are many directions for future work. First of all, features could be added to the semantics. For example, floats [3], bit-fields, concurrency [28], const qualifiers, register storage, *etc.*

Despite the fact that the biggest part of our interpreter has been implemented in Coq, it still uses some glue that is written in OCaml. In particular, the translation of string literals and `printf` (that is currently just present for debugging purposes) is inaccurate. To implement these features properly, we have to support functions on a variable number of arguments, I/O, and the const qualifier in CH_2O core C. Instead of supporting just basic forms of I/O such as `printf`, it may better to support external function calls in a general way. The work of Beringer *et al.* [1] may be useful for that.

Instead of using the CIL parser, which is written in OCaml, it would be interesting to use the parser by Jourdan *et al.* [12] that is part of CompCert and has been implemented and verified in Coq. This way, the entire interpreter could be implemented in Coq.

We have tested our semantics on a small test suite. Considering the bugs that we have found this way were minor and easy to fix, it seems to indicate that using a proof assistant to prove metatheory about the language [13–15, 18] has already provided a good way of debugging the semantics. Nonetheless, it would be useful to test the semantics with a more extensive test suite, like the GCC torture tests [8], or using a tool like CSmith [30].

Lastly, our implementation of the executable semantics implements the operational semantics in a rather naive way, and is using inefficient data structures to represent the memory (for example, it uses lists to represent array objects). In order to make the interpreter more robust, it would be useful to optimize the executable semantics. This could be achieved by using more efficient data structures, or by contracting more redexes in one step.

Acknowledgments

We thank Herman Geuvers and the anonymous reviewers for their helpful comments. This work is financed by the Netherlands Organisation for Scientific Research (NWO).

References

[1] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified Compilation for Shared-Memory C. In *ESOP*, volume 8410 of *LNCS*, pages 107–127, 2014.

[2] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaSript specification. In *POPL*, pages 87–100, 2014.

[3] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *ARITH*, pages 107–115, 2013.

[4] B. Campbell. An Executable Semantics for CompCert C. In *CPP*, volume 7679 of *LNCS*, pages 60–75, 2012.

[5] W. Dietz, P. Li, J. Regehr, and V. S. Adve. Understanding integer overflow in C/C++. In *ICSE*, pages 760–770, 2012.

[6] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[7] M. Felleisen, D. P. Friedman, E. E. Kohlbecker, and B. F. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[8] GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`.

[9] G. P. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[10] ISO. WG14 Defect Report Summary. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/`.

[11] ISO. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.

[12] J. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) Parsers. In *ESOP*, volume 7211 of *LNCS*, pages 397–416, 2012.

[13] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*, 2013.

[14] R. Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *POPL*, pages 101–112, 2014.

[15] R. Krebbers. Separation algebras for C verification in Coq. In *VSTTE*, volume 8471 of *LNCS*, 2014.

[16] R. Krebbers, X. Leroy, and F. Wiedijk. Formal C semantics: CompCert and the C standard. In *ITP*, volume 8558 of *LNAI*, pages 543–548, 2014.

[17] R. Krebbers and F. Wiedijk. A Formalization of the C99 Standard in HOL, Isabelle and Coq. In *CICM*, volume 6824 of *LNAI*, pages 297–299, 2011.

[18] R. Krebbers and F. Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS*, volume 7794 of *LNCS*, pages 257–272, 2013.

[19] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[20] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[21] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, 2012.

[22] P. Letouzey. A New Extraction for Coq. In *TYPES'02*, volume 2646 of *LNCS*, pages 200–219, 2003.

[23] A. Lochbihler and L. Bulwahn. Animating the Formalised Semantics of a Java-Like Language. In *ITP*, volume 6898 of *LNCS*, pages 216–232, 2011.

[24] N. Maclaren. What is an Object in C Terms?, 2001. Mailing list message, `http://www.open-std.org/jtc1/sc22/wg14/9350`.

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, volume 2304 of *LNCS*, pages 213–228, 2002.

[26] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[27] S. Owens, P. Böhm, F. Z. Nardelli, and P. Sewell. Lem: A Lightweight Tool for Heavyweight Semantics. In *ITP*, volume 6898 of *LNCS*, pages 363–369, 2011.

[28] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM*, 60(3):22, 2013.

[29] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: effective tool support for the working semanticist. In *ICFP*, pages 1–12, 2007.

[30] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.

[31] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, pages 427–440, 2012.