HAL
open science

A Precise and Abstract Memory Model for C Using Symbolic Values

Frédéric Besson, Sandrine Blazy, Pierre Wilke

To cite this version:

Frédéric Besson, Sandrine Blazy, Pierre Wilke. A Precise and Abstract Memory Model for C Using Symbolic Values. 12th Asian Symposium on Programming Languages and Systems (APLAS 2014), 2014, Singapore, Singapore. pp.449 - 468, 10.1007/978-3-319-12736-1_24 . hal-01093312

HAL Id: hal-01093312

https://inria.hal.science/hal-01093312v1

Submitted on 10 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Precise and Abstract Memory Model for C using Symbolic Values *

Frédéric Besson[1], Sandrine Blazy[2], and Pierre Wilke[2]

[1] Inria, Rennes, France
[2] Irisa - Université Rennes 1, Rennes, France

Abstract. Real life C programs are often written using C dialects which, for the ISO C standard, have undefined behaviours. In particular, according to the ISO C standard, reading an uninitialised variable has an undefined behaviour and low-level pointer operations are implementation defined. We propose a formal semantics which gives a well-defined meaning to those behaviours for the C dialect of the CompCert compiler. Our semantics builds upon a novel memory model leveraging a notion of symbolic values. Symbolic values are used by the semantics to delay the evaluation of operations and are normalised lazily to genuine values when needed. We show that the most precise normalisation is computable and that a slightly relaxed normalisation can be efficiently implemented using an SMT solver. The semantics is executable and our experiments show that the enhancements of our semantics are mandatory to give a meaning to low-levels idioms such as those found in the allocation functions of a C standard library.

1 Introduction

Semantics of programming languages give a formal basis for reasoning about the behaviours of programs. In particular, the correctness guarantee of C compilers [14] and verification frameworks [4] is stated with respect to the program semantics. However, the C programming language is specified in such a way that certain operations are either undefined, unspecified or implementation-defined. Typically, reading uninitialised memory is an undefined behaviour; the order of evaluation of function arguments is unspecified; the size of the `int` type is implementation-defined. A C program is *strictly conforming* if it does not trigger any undefined, unspecified or implementation-defined behaviour.

This leads to an unsettling question: what is the guarantee provided by a compiler when the program is not strictly conforming, *i.e.* when its semantics is undefined? The short answer is none. The C standard [10] explains that anything can happen with undefined behaviours: the compiler may fail to compile, but it could – and usually does – produce an executable code. The behaviour of the executable depends on compiler flags, especially optimisation levels. The executable code may behave as expected by the programmer, but it can also ignore

*This work was supported by Agence Nationale de la Recherche, grant number ANR-12-INSE-002 BinSec.

the statements that lead to undefined behaviours, or even crash at runtime. For instance, for the sake of optimisation, the compiler may choose to remove pieces of code that result in an undefined behaviour [19]. In summary, the advantage of undefined behaviours is that they can be exploited by C compilers to optimise the generated code; the downside is that C programs with undefined behaviours may have unexpected results.

In practice, undefined behaviours have been responsible for serious flaws in major open source software [19] – optimisations triggered by undefined behaviours have introduced vulnerabilities in the target code. Moreover, some low-level idioms cannot be expressed without resorting to unspecified behaviours of the C semantics. A compelling example is the memory allocation primitives of the C standard library which are written using the C syntax but do not have a *strictly conforming* semantics. One reason for this is that the low-level bit manipulation of pointers that is necessary for efficient and robust implementation of memory allocation is implementation defined.

To alleviate the problem, a common approach consists in setting compiler flags to disable optimisations known to exploit undefined behaviours [19, Section 3.1]. In a sense, flag tweaking is a fragile way to get the desired program semantics. Wang, Zeldovich *et al.* [20] propose a more principled compiler approach where they identify and report code whose optimisation depends on undefined behaviours. In this work, we advocate for a semantics-based approach and propose an executable extension of a C semantics ruling out unspecified behaviours originating from low-level pointer arithmetic and undefined behaviours due to access to uninitialised data.

The C standard describes only an informal semantics, but several realistic C formal semantics have been defined [17,3,7,12]. They describe precisely the defined behaviours of C programs, as well as some undefined behaviours. Yet, none of them accommodates for all low-level pointer manipulations; uninitialised data are only dealt with in a very limited fashion by the semantics of Ellison and Roşu [7, 6.2.2].

One formal semantics is the C semantics used by the CompCert formally verified C compiler [14]. CompCert is equipped with a machine-checked correctness proof establishing that the generated assembly code behaves exactly as prescribed by the semantics of the C source, eliminating all possibilities of compiler-introduced bugs and generating unprecedented confidence in this compiler. Yet, as for any compiler, the guarantee offered by CompCert only holds for programs with a defined behaviour. In general, the CompCert compiler provides a stronger guarantee than an ISO C compiler because its source language CompCert C is more defined than the ISO C99 language. Our goal is to extend the semantics expressiveness of CompCert C further by ruling out more undefined or unspecified behaviours.

The contributions of this work can be phrased as follows:

- We present the first formal semantics for CompCert C able to give a meaning to low-level idioms (bit-level pointer arithmetic and access to uninitialised data) without resorting to a concrete representation of the memory.

- The semantics operates over a novel memory model where *symbolic* values represent delayed computations and are normalised lazily to *concrete* values.
- We demonstrate that the most precise normalisation is decidable and explain how to devise an efficient implementation using an SMT solver.
- The semantics is executable and the software development is available at `http://www.irisa.fr/celtique/ext/csem/`.
- We show in our experiments that our extensions are mandatory to give a defined meaning to low-level idioms found in existing C code.

The remainder of this paper is organised as follows. Section 2 introduces relevant examples of programs having undefined or unspecified behaviours. Section 3 defines our extension of CompCert's semantics with symbolic values. Section 4 specifies our normalisation of symbolic values. Section 5 deals with the implementation of the normalisation using SMT solvers over the theory of bitvectors. Section 6 describes the experimental evaluation of our implementation. Related work is discussed in section 7, followed by concluding remarks.

2 Motivating Examples

An example of unspecified behaviour is the order of evaluation of the arguments of a function call. The relative size of numeric types is defined but the precise number of bits is implementation defined. An undefined behaviour is for instance the access of an array outside its bounds. Unsafe programming languages like C have by nature undefined behaviours and there is no way to give a meaningful semantics to an out-of-bound array access.[1] Yet, certain undefined behaviours of C were introduced on purpose to ease either the portability of the language across platforms or the development of efficient compilers. As illustrated below, our novel memory model gives a formal semantics to low-level idioms such as access to uninitialised memory or low-level pointer arithmetic.

2.1 Access to Uninitialised Variables

The C standard states that any read access to uninitialised memory triggers undefined behaviours [10, section 6.7.8, §10]: "If an object that has automatic storage duration is not initialised explicitly, its value is indeterminate." Here, "indeterminate" means that the behaviour is undefined. To illustrate a benefit of our semantics, consider the code snippet of Fig. 1, representative of an existing C pattern (see Section 6.3).

The program declares a `status` variable and sets its least significant bit using the `set` function. It then tests whether the least significant bit is set using the `isset` function. According to the C standard, this program has undefined behaviour because the `set` function reads the value of the `status` variable before it is ever written.

[1] Typed languages detect illegal accesses and typically throw an exception.

However, we argue that this program should have a well-defined semantics and should always return the value 1. The argument is that whatever the initial value of the variable `status`, the least significant bit of `status` is known to be 1 after the call `set(status,0)`. Moreover, the value of the other bits is irrelevant for the return value of the call `isset(status,0)` which returns 1 if and only if the least significant bit of the variable `status` is 1. More formally, the program should return the value of the expression `(status|(1≪0))&(1≪0) != 0` which evaluates to 1 whatever the value of `status`. Our semantics constructs symbolic values and normalises them to a genuine value when the evaluation yields a unique possible value.

2.2 Low-level Pointer Arithmetic

In ISO C, the bit width and the alignment of pointers are implementation defined. We consider here that pointers are encoded with 4 bytes and that the `malloc` function returns pointers that are 16-byte aligned (i.e. the 4 least significant bits are zeros). The C standard also states that arithmetic operations on pointers are limited to certain comparisons, the addition (or subtraction) of an integer offset to a pointer and the subtraction of two pointers pointing to the same object. In order to perform arbitrary operations over a pointer, it is possible to cast it to an unsigned integer of type **`uintptr_t`** for which the ISO C standard provides the following specification [10, Section 7.18.1.4].

> [The type **`uintptr_t`**] designates an unsigned integer type with the property that any valid pointer to **`void`** can be converted to this type, then converted back to pointer to **`void`**, and the result will compare equal to the original pointer.

Note that this specification is very weak and does not ensure anything if a pointer, cast to **`uintptr_t`**, is modified before being cast back. Here, **`uintptr_t`** is implemented by a 4 bytes unsigned integer, and a cast between pointers and **`uintptr_t`** preserves the binary representation of both pointers and integers (i.e. it is essentially a no-op).

With these assumptions in mind, consider the expected behaviour of the code snippet of Fig. 2. The pointer `p` is a 16-byte aligned pointer to a heap-allocated integer obtained by `malloc`. Therefore, it has 4 trailing spare bits. The pointer `q` is obtained from the pointer `p` by filling the 4 trailing bits (hence the bitwise `and` with `0xF`) with a hash of the pointer `p`. Note that this pattern is used in practice as a hardening technique to enforce pointer integrity [13]. Then, the evaluation

```
int set(int p, int flag) { return p | (1 ≪ flag); }
int isset(int p, int flag) { return (p & (1 ≪ flag)) != 0; }
int main() {  int status = set(status,0); return isset(status,0);   }
```

Fig. 1. Undefined behaviour: reading the uninitialised variable `status`

```
char hash(void* ptr);
int main(){
  int * p = (int *) malloc(sizeof(int));
  *p = 0;
  int * q = (int *) ((uintptr_t) p | (hash(p) & 0xF));
  int * r = (int *) (((uintptr_t) q >> 4) << 4);
  return *r;    }
```

Fig. 2. Unspecified behaviour: low-level pointer arithmetic

of `r` clears out the 4 trailing bits of `q` using logical shifts. We argue that `r` is equal to `p` and that the program has a well-defined semantics and returns `*p` (that is `0`). Our semantics computes the expected behaviour of this program without resorting to a concrete representation of pointers as machine integers.

2.3 Summary of Differences with the ISO C Standard

The ISO C standard leaves certain behaviours implementation defined. Among these, our semantics is parametrised by the size of the pointers and the alignment constraint of `malloc`. Our semantics also stipulates that pointer and `uintptr_t` types have the same size and that casts between these types preserve the binary representation of the objects. The ISO C standard states that reading uninitialised memory is undefined behaviour. Our semantics is more flexible and simulates the read of an arbitrary value. Operationally, our semantics propagates a symbolic undefined value through the execution. These extensions are sufficient to give a well-defined (and intuitive) semantics to the previous examples.

3 A C Semantics with Symbolic Values

Our semantics is able to model low-level idioms, in particular bit-level manipulation of pointers, while retaining abstraction properties of the current block-based memory model of CompCert. Our approach consists in computing symbolic values *lazily* delaying evaluation until values are really needed. A symbolic value sv evaluates to a value v if for every possible concrete memory M, sv evaluates to the same value v.

CompCert defines the semantics of a dozen of intermediate languages ranging from CompCert C to assembly. All the languages share the same memory model. The compiler transforms programs from one language to another and proves the correctness of the transformations with respect to that memory model. The two highest-level languages are CompCert C (source language) and Clight, a simpler version of CompCert C with side-effect free expressions. For the sake of presentation, we introduce our new memory model on the Clight semantics. However, our implementation leverages the existing CompCert C interpreter enhanced with our new memory model.

In this section, we first describe our memory model with symbolic values. Then, we show how to enhance the Clight semantics [3] with symbolic values.

Memory locations:	l ::= (b, i)	(block, integer offset)
Values:	v ::= `int`(i) | `float`(f) | `ptr`(l) | `undef`	
Memory chunks:	κ ::= `Mint8signed`	8-bit integers
	| `Mint8unsigned`	
	| `Mint16signed`	16-bit integers
	| `Mint16unsigned`	
	| `Mint32`	32-bit integers or pointers
	| `Mfloat32`	32-bit floats

Operations over memory states:

`alloc`$(M, lo, hi) = (M', b)$	Allocate a fresh block with bounds $[lo, hi[$.
`free`$(M, b) = M'$	Free (invalidate) the block b
`load`$(\kappa, M, b, i) = \lfloor v \rfloor$	Read consecutive bytes (as determined by κ) at block b, offset i of memory state M. If successful, return the contents of these bytes as value v.
`store`$(\kappa, M, b, i, v) = \lfloor M' \rfloor$	Store the value v as one or several consecutive bytes (as determined by κ) at offset i of block b. If successful, return an updated memory state M'.
`bound`(M, b)	Return the bounds $[lo, hi[$ of block b.

Fig. 3. CompCert's memory model

3.1 The CompCert Memory Model

The semantics of operations involving pointers relies on a memory model defining how values are represented. The most concrete memory model is an array of bytes, where pointers and integers are indistinguishable. It can give a precise semantics, but reasoning on programs at such a low level is cumbersome (e.g. reasoning on forbidden memory accesses to detect buffer overflows). CompCert is using a more abstract block-based model [15] where memory is divided into disjoint blocks, each block corresponding to an allocated variable. A memory is a collection of blocks, each block being an array of concrete bytes. Intuitively, a block represents a C variable or an invocation of `malloc`.

Values stored in memory are defined in Fig. 3. They are the disjoint union of 32-bit integers (written as `int(i)`), 32-bit floating-point numbers (written as `float(f)`), locations (written as `ptr(b,i)`), and the special value `undef` representing the content of uninitialised memory. Locations `ptr(b,i)` are composed of a block identifier `b` (i.e. an abstract address) and an integer byte offset `i` within this block. Pointer arithmetic modifies the offset part of a location, keeping its block identifier part unchanged. Memory chunks appear in memory operations `load` and `store`, to describe concisely the size, type and signedness of the value being stored. These functions return option types: we write $\emptyset$ for failure and $\lfloor x \rfloor$ for a successful return of a value x.

The memory is modelled as a map associating to each location an 8-bit elementary memory value of type `memval`. A `memval` value is a byte-sized quantity that describes the current content of a memory cell. It can be either `Undef` to model uninitialised memory; $\texttt{Byte}(b)$ to model a concrete byte b; or $\texttt{Pointer}(b, i, n)$ to represent the n-th byte ($n \in \{1, 2, 3, 4\}$) of the location $\texttt{ptr}(b, i)$.

3.2 A New Memory Model with Symbolic Values

Our memory model is built on top of CompCert's, where we replace the values with symbolic values, defined as follows.

$$
\begin{array}{rll}
sv ::= & v & \\
| & \texttt{op}_1^{\tau}\ sv & \text{unary arithmetic operation} \\
| & sv\ {}^{\tau}\texttt{op}_2{}^{\tau}\ sv & \text{binary arithmetic operation} \\
| & sv^{\tau}\ \texttt{?}\ sv^{\tau}\ \texttt{:}\ sv^{\tau} & \text{conditional expression} \\
| & (\tau)sv^{\tau} & \text{type cast (to a C type } \tau)
\end{array}
$$

Symbolic values are annotated by C types that are needed to disambiguate overloaded C operators. Symbolic values are side-effect free and therefore their evaluation is independent of the memory content. To account for alignment properties we associate with each block of the memory a *mask* that the concrete address of the block needs to satisfy. Formally, the concrete address a of a block with mask msk must be such that $a \,\&\, msk = a$. The allocation primitive is modified accordingly, and we add a function that returns the mask of a given block. The primitives `load` and `store` now operate over symbolic values instead of values.

$$
\begin{array}{l}
\texttt{alloc}(M, lo, hi, msk) = (M', b) \\
\texttt{mask}(M, b) = msk \\
\texttt{load}(\kappa, M, b, i) = \lfloor sv \rfloor \\
\texttt{store}(\kappa, M, b, i, sv) = \lfloor M' \rfloor
\end{array}
$$

We also adapt the `memval` type to accommodate for symbolic values. To that purpose, we replace the `Pointer` constructor by a generalised $\texttt{Symbolic}(sv, n)$ constructor which represents the n-th byte of a symbolic value sv.

To perform these memory primitives, we define a key operation **`extr`**`(sv,i)`, which extracts the ith byte of a symbolic value. The reverse operation is the concatenation of a symbolic value `sv1` with a symbolic value `sv2` representing 8 bits. Assuming that the symbolic value represents a 32-bit value, these operations can be defined as

```
extr(sv,i) = (sv ≫(8*i)) & 0xFF
concat(sv1,sv2) = sv1 ≪8 + sv2
```

3.3 Parametrised Semantics of Clight Values

Expressions cannot be kept symbolic forever. Our semantics is equipped with a partial normalisation function $\texttt{normalise}(M, \tau, sv)$ which converts a symbolic

Expressions in l-value position:

$$\frac{G, E \vdash a, M \Rightarrow sv \quad \texttt{normalise}(M, \texttt{type}(a), sv) = \lfloor \texttt{ptr}(\ell) \rfloor}{G, E \vdash *a, M \Leftarrow \ell} \quad (1)$$

Expressions in r-value position:

$$\frac{G, E \vdash a_1, M \Rightarrow sv_1 \quad \texttt{type}(a_1) = \tau}{G, E \vdash op_1\ a_1, M \Rightarrow op_1^{\tau}\ sv_1} \quad (2)$$

$$\frac{G, E \vdash a_1, M \Rightarrow sv_1 \quad G, E \vdash a_2, M_1 \Rightarrow sv_2 \quad \texttt{type}(a_1) = \tau_1 \quad \texttt{type}(a_2) = \tau_2}{G, E \vdash a_1\ op_2\ a_2, M \Rightarrow sv_1\ {}^{\tau_1}op_2{}^{\tau_2}\ sv_2} \quad (3)$$

$$\frac{\begin{array}{c} G, E \vdash a_1, M \Rightarrow sv_1 \quad \texttt{type}(a_1) = \tau_1 \\ G, E \vdash a_2, M \Rightarrow sv_2 \quad \texttt{type}(a_2) = \tau_2 \quad G, E \vdash a_3, M \Rightarrow sv_3 \quad \texttt{type}(a_3) = \tau_3 \end{array}}{G, E \vdash a_1\ \texttt{?}\ a_2\ \texttt{:}\ a_3, M \Rightarrow sv_1^{\tau_1}\ \texttt{?}\ sv_2^{\tau_2}\ \texttt{:}\ sv_3^{\tau_3}} \quad (4)$$

$$\frac{G, E \vdash a, M \Rightarrow sv_1 \quad \texttt{type}(a) = \tau_1}{G, E \vdash (\tau)a, M \Rightarrow (\tau)sv_1^{\tau_1}} \quad (5)$$

Fig. 4. Semantics of Clight with symbolic values (excerpt)

value sv to a concrete value of type τ, depending on masks and bounds of blocks of memory M. The modified Clight semantics of expressions is given in Fig. 4. It is defined by judgements, parametrised by a global environment G, a local environment E and an initial memory state M. The evaluation of an expression in l-value (resp. r-value) position results in a location (resp. symbolic value). In the judgements, a, a_1, a_2, a_3 range over syntactic expressions and sv, sv_1, sv_2, sv_3 range over symbolic values.

$G, E \vdash a, M \Leftarrow \ell$ (evaluation of an expression in l-value position)
$G, E \vdash a, M \Rightarrow sv$ (evaluation of an expression in r-value position)

Compared to the existing Clight semantics [3], expressions are not completely evaluated but mapped to symbolic values. Moreover, the rules explicitly introduce calls to a normalisation function (see Section 4). These calls are inserted when a genuine value is required, *i.e.* when reading from or writing to memory, when evaluating the condition of a loop or `if-then-else` statement, or when exiting the program.

For instance, to evaluate $*a$, rule (1) recursively evaluates the expression a to get the symbolic value sv. To get a genuine location l, rule (1) explicitly normalises sv to get l. Now, l can be used to perform a `store` memory operation. Rule (2) specifies the evaluation of unary expression $op_1\ a_1$: it recursively evaluates the expression a_1 to get the symbolic value sv_1. Instead of evaluating the operator op_1, the semantics delays the evaluation and constructs the symbolic value $op_1^{\tau}\ sv_1$ where τ is the type of the expression a_1. Similarly, the evaluation of binary expressions (rule (3)), conditional expressions (rule (4)) and cast

expressions (rule (5)) recursively evaluate their arguments and construct a symbolic value. Note that for the original Clight semantics, two rules are needed to give a semantics to conditional expressions [3, Fig.6, rules (12) and (13)] depending on whether the condition holds or not. With symbolic values, we delay the evaluation and therefore have a single rule.

4 A Sound and Complete Normalisation

Our semantics with symbolic values aims at giving a defined meaning to low-level idioms that are out-of-reach of the current Clight. To do so, we need to instantiate the semantics with an aggressive normalisation function. The existing Clight semantics can be obtained by a suitable normalisation function. This semantics is trustworthy because it has been carefully designed, thoroughly reviewed and intensively tested. However, for more aggressive normalisation (which is what we aim at), this validation methodology does not scale and therefore provides a limited trustworthiness.

In this section, we give a formal specification of the `normalise` function. We define the notions of soundness and completeness of a normalisation function with respect to a concrete memory model. We will later show (Section 5) how to get efficient executable implementations from this specification.

4.1 Soundness of the Normalisation of Symbolic Values

Our semantics is parametrised by the normalisation function `normalise`. In this part, we describe the soundness conditions that this normalisation should fulfil. Symbolic values denote low-level values of types either `Tint` or `Tfloat`. The mapping between high-level C types and low-level types is performed by the function `ctyp` defined as follows: $\texttt{ctyp}(\tau) = \texttt{Tfloat}$ if $\tau = \texttt{Tfloat}$, and $\texttt{ctyp}(\tau) = \texttt{Tint}$ otherwise. Notice that all the pointer types are mapped to the type `Tint`. Indeed, at low-level, addresses are not distinguishable from genuine integers. To map locations (b, i) to integers, the low-level evaluation is equipped with a mapping A from block identifiers to concrete addresses, which assigns an address to each memory block and therefore fixes a memory layout. In general, the low-level evaluation of a symbolic value is not a single value but a set because the value `undef` represents an arbitrary low-level value. Definition 1 formalises the low-level evaluation of symbolic values.

Definition 1 (Low-level evaluation). *Let A be an allocation function mapping block identifiers to concrete addresses. The low-level evaluation $[\![\cdot]\!]^{\tau}_{A}$ of a symbolic value e of type τ is inductively defined by the following rules.*

$$\frac{}{\texttt{int}(n) \in [\![\texttt{int}(n)]\!]^{\texttt{Tint}}_{A}} \qquad \frac{}{\texttt{float}(f) \in [\![\texttt{float}(f)]\!]^{\texttt{Tfloat}}_{A}}$$

$$\frac{\texttt{ctyp}(\tau) = \texttt{Tint}}{\texttt{int}(A(b) + i) \in [\![\texttt{ptr}(b, i)]\!]^{\tau}_{A}} \qquad \frac{\texttt{ctyp}(\tau) = \texttt{Tfloat}}{\texttt{float}(n) \in [\![\texttt{undef}]\!]^{\tau}_{A}}$$

$$\frac{\texttt{ctyp}(\tau) = \texttt{Tint}}{\texttt{int}(n) \in [\![\texttt{undef}]\!]_A^\tau}$$

$$\frac{v_1 \in [\![sv_1]\!]_A^{\tau_1} \quad \texttt{eval_unop}(op_1, v_1, \texttt{ctyp}(\tau_1)) = \lfloor v \rfloor \quad \texttt{type}(v) = \texttt{ctyp}(\tau)}{v \in [\![op_1^{\tau_1}\ sv_1]\!]_A^\tau}$$

$$\frac{\begin{array}{c} v_1 \in [\![sv_1]\!]_A^{\tau_1} \quad v_2 \in [\![sv_2]\!]_A^{\tau_2} \quad \texttt{type}(v) = \texttt{ctyp}(\tau) \\ \texttt{eval_binop}(op_2, v_1, \texttt{ctyp}(\tau_1), v_2, \texttt{ctyp}(\tau_2)) = \lfloor v \rfloor \end{array}}{v \in [\![sv_1\ {}^{\tau_1} op_2{}^{\tau_2}\ sv_2]\!]_A^\tau}$$

$$\frac{v_1 \in [\![sv_1]\!]_A^{\tau_1} \quad v_2 \in [\![sv_2]\!]_A^{\tau} \quad \texttt{is_true}(v_1, \texttt{ctyp}(\tau_1))}{v_2 \in [\![sv_1^{\tau_1}\ \texttt{?}\ sv_2^\tau\ \texttt{:}\ sv_3^\tau]\!]_A^\tau}$$

$$\frac{v_1 \in [\![sv_1]\!]_A^{\tau_1} \quad v_3 \in [\![sv_3]\!]_A^{\tau} \quad \texttt{is_false}(v_1, \texttt{ctyp}(\tau_1))}{v_3 \in [\![sv_1^{\tau_1}\ \texttt{?}\ sv_2^\tau\ \texttt{:}\ sv_3^\tau]\!]_A^\tau}$$

$$\frac{v_1 \in [\![sv_1]\!]_A^{\tau_1} \quad \texttt{cast}(v_1, \texttt{ctyp}(\tau)) = \lfloor v \rfloor}{v \in [\![(\tau) sv_1^{\tau_1}]\!]_A^\tau}$$

By construction, the denotation of a symbolic value of type τ is a set of values of type $\texttt{ctyp}(\tau)$. Symbolic values that are not well-typed have an empty low-level evaluation. As a side-remark, notice that the types of symbolic values cannot be uniquely inferred (`undef` can have an arbitrary type), and are therefore explicitly given. Moreover, the low-level evaluation of a symbolic value is reusing the existing high-level operators `eval_unop` and `eval_binop` with the difference that types are low-level types.

The normalisation of a symbolic value s should return a defined value v ($v \neq \texttt{undef}$) such that s evaluates to v for all possible concrete valid memory layouts. Definition 2 specifies valid memory layouts.

Definition 2 (Valid memory layout). *An allocation A from blocks to concrete addresses is a valid memory layout for memory M (written $A \vdash M$) iff:*

1. *addresses from distinct blocks do not overlap,*
2. *the address of a block satisfies its mask, i.e.* $\forall b, A(b)\ \&\ \texttt{mask}(M, b) = A(b)$
3. *addresses are not equal to zero.*

With the previous definitions we are ready to state what it means for a normalisation to be sound.

Definition 3 (Sound normalisation). *A normalisation function is sound iff for any symbolic value sv, it returns a value v ($\texttt{normalise}(M, \tau, sv) = \lfloor v \rfloor$) such that v is not* `undef` *; v has type τ and v has the same evaluation as sv for any valid allocation layout i.e.* $\forall A \vdash M. [\![sv]\!]_A^\tau = [\![v]\!]_A^\tau$.

Note that because v differs from `undef`, $[\![v]\!]_A^\tau$ is necessarily a singleton. Yet, certain symbolic values containing `undef` can nonetheless be normalised, for instance, `normalise(M,Tint,undef & 0x0)`$= \lfloor 0 \rfloor$.

4.2 Reconstructing the Original Clight Semantics

The more precise the normalisation, the more defined the semantics. There is a hierarchy of normalisations of different precision. We therefore aim at identifying a normalisation which is not only precise but also tractable. The least precise normalisation, which always returns $\emptyset$, is sound but useless: it fails to provide a semantics to any expression. The original Clight semantics can be modelled by a normalisation function which recursively evaluates symbolic values.

$$\frac{\texttt{type}(v) = \tau}{\texttt{normalise}(M, \tau, v) = \lfloor v \rfloor}$$

$$\frac{\texttt{normalise}(M, \tau_1, e_1) = \lfloor v_1 \rfloor \quad \texttt{eval_unop}(op_1, v_1, \tau_1) = \lfloor v \rfloor \quad \texttt{type}(v) = \tau}{\texttt{normalise}(M, \tau, \texttt{op}_1^{\tau_1}\ e_1) = \lfloor v \rfloor}$$

$$\frac{\begin{array}{c}\texttt{normalise}(M, \tau_1, e_1) = \lfloor v_1 \rfloor \quad \texttt{normalise}(M, \tau_2, e_2) = \lfloor v_2 \rfloor \\ \texttt{eval_binop}(op_2, v_1, \tau_1, v_2, \tau_2) = \lfloor v \rfloor \quad \texttt{type}(v) = \tau\end{array}}{\texttt{normalise}(M, \tau, e_1\ {}^{\tau_1}\texttt{op}_2{}^{\tau_2}\ e_2) = \lfloor v \rfloor}$$

As explained in the introduction, this normalisation is unable to give a semantics to low-level pointer operations (e.g. `ptr(b,i) & 0x0`) or expressions with undefined sub-terms (e.g. `undef & 0x0`). The original Clight semantics could be enriched to cope with these simple expressions. However, dealing with arbitrarily complex expressions using *ad hoc* simplifications would not be manageable.

4.3 Completeness of the Normalisation of Symbolic Values

Whenever possible, the most precise normalisation should always return some value. Yet there are rare cases where distinct values are sound normalisations. This is illustrated by Example 1.

Example 1. Consider the normalisation of the symbolic value $\texttt{ptr}(b, 0) + 2^{31} - 1$ for a pointer type τ in a memory M made of an unaligned block b with bounds $[0, 2^{31} - 1[$ and a 2-byte aligned block b' with bounds $[0, 2^{31}[$. Because the maximum capacity of the memory is $2^{32} - 1$ bytes (0 is not a valid address), the memory is full. Moreover, the alignment constraint of b' prevents it from being allocated at address 1. It follows that the only valid memory layout of M is $A = [b \mapsto 1; b' \mapsto 2^{31}]$.

As we have $[\![\texttt{ptr}(b, 0) + 2^{31} - 1]\!]_A^\tau = [\![\texttt{ptr}(b, 2^{31} - 1)]\!]_A^\tau = [\![\texttt{ptr}(b', 0)]\!]_A^\tau = \{\texttt{int}(2^{31})\}$, both $\texttt{ptr}(b, 2^{31} - 1)$ and $\texttt{ptr}(b', 0)$ are sound normalisations.

In this example, the normalisation $\texttt{ptr}(b', 0)$ is more valuable because it represents a valid address (i.e. within the bounds of the allocated block). Definition 4 formalises what it means for a normalisation to be complete by stipulating an ordering $\prec_M$ on values such that $\texttt{ptr}(b', 0) \prec_M \texttt{ptr}(b, 2^{31} - 1)$.

Definition 4 (Complete normalisation). *A normalisation function* `norm` *is complete if for all sound normalisations* $\texttt{norm}'$*, we have:*

$$\texttt{norm}(M, \tau, e) \preccurlyeq_M \texttt{norm}'(M, \tau, e)$$

where $\preccurlyeq_M$ is the reflexive closure of the ordering $\prec_M$ inductively defined below and $<$ is an arbitrary total order over locations.

$$\lfloor v \rfloor \prec_M \emptyset \quad (6) \qquad \frac{i \in \texttt{bound}(M, b) \quad i' \notin \texttt{bound}(M, b')}{\lfloor \texttt{ptr}(b, i) \rfloor \prec_M \lfloor \texttt{ptr}(b', i') \rfloor} \quad (7)$$

$$\frac{i \in \texttt{bound}(M, b) \quad i' \in \texttt{bound}(M, b') \quad (b, i) < (b', i')}{\lfloor \texttt{ptr}(b, i) \rfloor \prec_M \lfloor \texttt{ptr}(b', i') \rfloor} \quad (8)$$

$$\frac{i \notin \texttt{bound}(M, b) \quad i' \notin \texttt{bound}(M, b') \quad (b, i) < (b', i')}{\lfloor \texttt{ptr}(b, i) \rfloor \prec_M \lfloor \texttt{ptr}(b', i') \rfloor} \quad (9)$$

Rule (6) ensures that a complete normalisation is maximally defined and as much as possible does not return $\emptyset$. Rules (7), (8) and (9) ensure that a complete normalisation should, as much as possible, return a valid address.

There are memories M for which there is no valid memory layout A. The simple case is when the size of the allocated memory exceeds $2^{32}-1$ bytes. In general, reasoning about the size of the allocated memory is not enough because the memory can be fragmented due to alignment constraints. In such cases, Definition 4 is not sufficient to ensure that there is a unique sound and complete normalisation function. The reason is that when there is no valid memory layout, *any* value v is a correct normalisation. Moreover, as the order $\prec_M$ is not total, Definition 4 does not rule out these spurious cases. The good news is that all sound and complete normalisations compute the same result as soon as there exists a valid memory layout. Our normalisation algorithm (see Section 5) checks the existence of a valid memory layout and fails to normalise when there is none.

5 Evaluating Symbolic Values using an SMT solver

We have adapted the CompCert C semantics and its executable interpreter to work with symbolic values. As already demonstrated for the Clight semantics, the addition of symbolic values is not very intrusive and reuses most of the semantics infrastructure of the existing interpreter.

The difficulty lies in the implementation of the normalisation function. Given a memory M, there are finitely many valid memory layouts A. It is thus decidable to compute a sound and complete normalisation and the naive algorithm consists in enumerating over the valid memory layouts and checking that the symbolic values always evaluate to the same values. Yet, this is not tractable. As shown below, the normalisation can be recast as a decision problem over the logic of bitvectors. However, implementing (and proving) in Coq an efficient decision procedure for this logic would require a substantial engineering effort. Therefore, our current implementation leverages an external Satisfiability Modulo Theory (SMT) solver, Z3 [6].

5.1 An Executable Semantics of Symbolic Values

We have adapted the CompCert C interpreter to work with symbolic values. The modification requires to change the type of values to the type of symbolic values and to replace the existing memory model by our implementation accommodating for symbolic values. As it is illustrated for Clight, the evaluation of C operators now builds symbolic values and calls to the **`normalise`** function are placed at certain points, as discussed below.

Our memory model stores (resp. reads) symbolic values to (resp. from) memory but the address needs to be a location `(b,i)`. Therefore, we apply the normalisation function before calling the **`store`** and **`load`** primitives of the memory model. The normalisation is also called to compute the target of conditional jumps (e.g. **`for`**, **`while`** or **`if`** statements). A last normalisation is applied before ending the program because the program status needs to be a genuine integer. If the normalisation succeeds and returns some value, then the execution continues normally. Otherwise, the semantics gets stuck and the interpreter returns that it encountered an undefined behaviour. We detail in Section 6 some representative programs of our benchmarks.

5.2 Normalisation as a Satisfiability Problem

The normalisation function is axiomatised and implemented by an external (trusted) call to the SMT solver Z3 [6]. As stated earlier, the problem of computing the most precise normalisation is decidable. Yet, a naive approach does not provide a tractable algorithm. A better solution consists in encoding the normalisation problem as an SMT problem over the logic of bitvectors and uninterpreted function symbols. A bitvector of size n is the logic counterpart of a machine integer with n bits. This logic is therefore a perfect match for reasoning about machine integers.

First, we *axiomatise* the memory and define a logical function *size* mapping each block to its size and a logical function *mask* mapping each block to the mask to be verified by the concrete address. Next, we axiomatise the valid memory layout relation by directly translating Definition 2 in first-order logic.

Example 2. Consider a memory M restricted to two blocks b_1 and b_2, with b_1 of bounds $[0, 4[$ aligned on word boundaries (i.e. the 4 trailing bits are zeros) and b_2 of bounds $[0, 8[$ with no alignment constraint. The axiomatisation of M

is given by the following formulae.

$$\begin{aligned}
\text{Block sizes:} \quad & \mathit{size}(b) = \begin{cases} 4 \text{ if } b = b_1 \\ 8 \text{ if } b = b_2 \\ 0 \text{ otherwise} \end{cases} \\
\text{Block masks:} \quad & \mathit{mask}(b) = \begin{cases} \texttt{0xFFFC} \text{ if } b = b_1 \\ \texttt{0xFFFF} \text{ if } b = b_2 \\ \texttt{0xFFFF} \text{ otherwise} \end{cases} \\
\text{No overlap:} \quad & \forall b, b', o, o'. \bigwedge \begin{cases} b \neq b' \\ o < \mathit{size}(b) \\ o' < \mathit{size}(b') \end{cases} \Rightarrow A(b) + o \neq A(b') + o' \\
\text{Addresses are not 0:} \quad & \forall b, o. o < \mathit{size}(b) \Rightarrow A(b) + o \neq 0 \\
\text{Alignment :} \quad & \forall b, A(b) \,\&\, \mathit{mask}(b) = A(b)
\end{aligned}$$

We process the symbolic value e to be normalised into a logical symbolic value e^* and replace occurrences of **`undef`** by distinct fresh logical variables thus modelling that **`undef`** may take any value.

Normalising into an integer. To normalise into an integer, we generate the SMT query: $e^* = i$, where i is a fresh logical variable. Suppose the formula is satisfiable for a value v for logical variable i. This means that there exists a valid memory layout such that e is evaluated as the value v. However, this value v is only a sound normalisation if it is the evaluation for *every* possible valid memory layout. To ensure this, we generate the second SMT query: $e^* = i \wedge i \neq v$. If this is unsatisfiable, then we will return v as the normalisation of e.

Normalising into a pointer. Getting the normalisation of a pointer value is more complicated by the fact that there are several ways of decomposing an integer into a location made of a base and an offset. Yet, as we are only interested in valid addresses (i.e. with an offset inside the bounds of the block), there is only a single choice. Therefore, we generate the following SMT query:

$$e^* = A(b) + o \wedge o < \mathit{size}(b).$$

Given a model (b', o') for location (b, o), we have to ensure that the evaluation of the expression is independent from the memory layout. Since blocks do not overlap, there is only one block such that the pointer is valid, so we just need to check that b' is the only possible block that makes a valid pointer, i.e. that the following formula is unsatisfiable:

$$e^* = A(b) + o \wedge o < \mathit{size}(b) \wedge b \neq b'$$

Example 3. Consider again the memory M of Example 2 and the symbolic value $e = \texttt{ptr}(b_1, 1) - \texttt{ptr}(b_2, 2) + \texttt{ptr}(b_2, 4) + \texttt{undef}\&\texttt{0x0}$. We process e into a logical expression e^* by replacing `undef` by the fresh variable x_1:

$$e^* = A(b_1) + 1 - A(b_2) - 2 + A(b_2) + 4 + x_1\&\texttt{0x0}$$

Notice that the two occurrences of $A(b_2)$ cancel out each other, and that we have $\forall x, x\&\texttt{0x0} = 0$. As a result, we can simplify this expression e^* into $A(b_1) + 3$.

Normalising into an integer. We need to solve the following SMT query, with i the unkown: $A(b_1) + 3 = i$. We then get a first solution (e.g. $v = 19$, with $A(b_1) = 16$). However, this is not the only possibility because we get a second solution with $A(b_1) = 32$ for example, which yields $v = 35$. This expression has therefore no normalisation as an integer.

Normalising into a pointer. Now, the SMT query we need to solve is:

$$A(b_1) + 3 = A(b) + o \wedge o < size(b)$$

A solution is $b' = b_1$ and $o' = 3$, and we can see that this is the only solution to this equation. Therefore the expression e is normalised into the location $\texttt{ptr}(b_1, 3)$.

5.3 Relaxation and Optimisation of the SMT Encoding

The previous encoding of the memory depends on the number of allocated blocks. Thus, as the memory gets bigger, the normalisation would get slower. In practice, we observe that the size of the memory has a dramatic (negative) impact on SMT solvers. To tackle the problem, we propose a relaxation of the SMT query that is independent of the number of allocated blocks and only depends on the size of the symbolic value to be normalised.

A key observation is that a symbolic value can only be normalised if the corresponding SMT query has a unique solution. As a result, it is always sound to relax the SMT query and generate a weaker one (i.e. with potentially more solutions) provided the initial formula is satisfiable. Indeed, if there are more solutions, the normalisation will fail – this is always sound.

In our relaxation, we do not fully axiomatise the memory but only specify the bounds and masks of the memory blocks B that appear syntactically in the symbolic value to be normalised. When normalising a symbolic pointer, we also state explicitly in the SMT query that the normalisation, if it exists, should be a location (b, i) such that $b \in B$.

This relaxation will only miss a normalisation if the memory is almost full and blocks $b \in B$ cannot be allocated at certain addresses because of bound or alignment constraints of other blocks $b' \notin B$. This is illustrated by Example 4.

Example 4. Consider a memory with 3 unaligned blocks b_1, b_2 and b_3 of size 1 and a last block b_4 of size $2^{32} - 4$ that is 4-byte-aligned, i.e. the last two bits are zeros. Because of alignment and size constraints, the block b_4 can only be allocated at address 4 while other blocks can be allocated at the remaining addresses (i.e. 1, 2 and 3). As a result, the symbolic value $\texttt{ptr}(b_1,0)+\texttt{ptr}(b_2,0)+\texttt{ptr}(b_3,0)$ evaluates to 6 which corresponds to the valid location $(b_4, 2)$.

The normalisation of Example 4 requires a full axiomatisation of the memory and cannot be obtained using our relaxation. In practice, we have never encountered such a pathological case.

6 Experimental Evaluation

As stated earlier, we have adapted the CompCert C interpreter so that we could test our semantics on real programs. This required only minor changes to get it to work with symbolic values. However, we put slightly more effort in designing stubs in the interpreter to model system calls such as `mmap` that are used e.g. in the source code of the `malloc` implementation we used. This system call is mapped to the `alloc` primitive of our memory model. Other system calls such as `open`, `read` or `write` are resolved using the OCaml equivalent functions.

We have tested our C semantics with symbolic values on the CompCert benchmarks. Their size ranges between a few hundreds and a few thousands lines of code. We checked the absence of regression: when the CompCert interpreter returns a defined value, our interpreter enhanced with symbolic values returns the exactly same value.

We have also run our interpreter over Doug Lea's memory allocator [13] and on parts of the NaCl cryptographic library [2], which are challenging programs because they perform low-level pointer arithmetic; their size is about a few thousands lines of code. For this experiment, we model the system call `mmap` by a call to the `alloc` primitive of our memory model with a mask specifying the alignment of a page. Our interpreter succeeds in giving a semantics to memory management functions, such as `malloc`, `memalign` or `free`, built on top of `mmap`. As there is no other formal C semantics able to deal with low-level pointer arithmetic, we checked that the result of our interpreter was matching the output of `gcc`. Programs reading uninitialised variables have undefined semantics and `gcc` could exploit this to perform arbitrary computations. Yet, the output of `gcc` and our interpreter agree on examples similar to Fig. 1. In the following, we detail some interesting patterns found in the benchmarks.

6.1 Pointer Arithmetic Using Alignment and Bitwise Operations

The `malloc` function sometimes needs to check a pointer's distance to an alignment boundary. This is equivalent to getting the last bits of the pointer. For instance, this is done with the C expression `p & 15`, which gets the 4 last bits of pointer `p`. For our experiments, pointers are allocated by `mmap` and are therefore known to be aligned on more than 16 bytes boundaries. For a pointer `p=`**`ptr`**`(b,3)`, our SMT encoding models that the last 4 bits of `b` are zeros and the code evaluates to `3&15` (i.e. `3`). In general, with the previous alignment constraints, we have that the symbolic value **`ptr`**`(b,o) & 15` returns the offset `o` of the pointer.

A similar example is the function `memalign(al,nb)`, where `al` must be a power of two (i.e. `al` $= 2^n$). The function dynamically allocates a `nb`-byte region, and ensures that the address returned has the n last bits to zero. When called with `al` $= 32$, the function computes checks such as `p&31 == 0` to check that the 5 last bits are zeros. The left-hand side of the comparison is evaluated in the same manner as the example above, and the comparison is computed trivially.

6.2 Comparison Between Pointers and -1

Several system calls, such as `mmap` or `sbrk`, are expected to return pointers but return -1 on error. When a function calls `mmap` for example, there is typically a check that the system call succeeded (i.e. the returned value is not -1).

```
void *p = mmap(...); if (p == -1) { ... }
```

Our normalisation gives a semantics to this programming pattern using the following reasoning. We know that pointers returned by `mmap` are aligned on a page boundary (2^{12} in our implementation, i.e. the 11 last bits of the pointer are zeros). When the allocation succeeds, the pointer can therefore never be -1 (in binary `0xFFFFFFFF`) thus allowing to evaluate this comparison.

6.3 Operations on Undefined Values

The example shown is Fig. 1 is a simplified version of a C expression that appears in real-life programs. For example, the `memalign` function described above features this kind of operations on undefined values.

The memory managed by the dynamic allocation functions is organised in memory chunks, which consist of two 32-bit words of meta-data and the memory chunk itself. The second word of meta-data stores the size of the chunk and two bits of other information. Initialising the meta-data is done with the C code `*p = (*p & 0b1)|size|0b10` (the `0b` prefix applies to constants in binary format). When the memory pointed by `p` is undefined, this ends up with the symbolic value `(undef & 0b1)|size|0b10`. It does not evaluate as a value, because the last bit is still undefined.

However, our semantics enables us to keep a symbolic value holding information about all the other bits instead of getting stuck. For instance, the symbolic value `((undef & 0b1)|size|0b10) & 0b10` has the well-defined normalisation `0b10` and retrieves the second last bit of the meta-data. This reasoning is made possible by the fact that `size` is a multiple of 4 (i.e. the last two trailing bits of `size` are zeros).

6.4 Copying Bytes between Memory Areas with `memmove`

Our semantics requires the target of jump instructions to be unique. This is a consequence of the fact that a symbolic value representing a conditional should normalise to some fixed boolean value. In other words, a program whose control-flow depends on the memory layout has an undefined behaviour. This dependance on the memory layout (e.g. on the memory allocator) is a portability bug that is detected by our semantics.

Indeed, in our experiments, we have encountered this situation for the `memmove` function (see Fig. 5) which implements a memory copy even when the origin and destination memory regions do overlap. To cope with this situation, the `memmove` function performs the pointer comparison `dest <= src`. If the pointers `dest` and `src` point to distinct memory blocks, this comparison depends on the memory layout and is therefore undefined for our memory model.

```
void * memmove( void * s1, const void * s2, size_t n ) {
  char * dest = (char *) s1;
  const char * src = (const char *) s2;
  if ( dest <= src )
     while ( n-- ) { *dest++ = *src++; }
  else {
    src += n; dest += n;
    while ( n-- ) { *--dest = *--src; }
  }
  return s1;
}
```

Fig. 5. `memmove` with an undefined semantics

We have solved the issue by replacing the original condition `dest <= src` with the more involved condition `src <= dest & dest < src + n`. This condition explicitly tests whether the memory regions overlap using the integer `n` which is the number of bytes to be copied. Notice that we use on purpose the bitwise `&` operator (and not the lazy boolean `&&` operator). A `&&` would force the evaluation of `src <= dest` which cannot be normalised. The new condition with a `&` constructs a symbolic value which is independent from the memory layout and has therfore always a defined normalisation. In particular, if the pointers are from distinct blocks, the condition is always false because locations from distinct blocks cannot overlap.

7 Related Work

Wang *et al.* have shown that undefined behaviours of the ISO C standard have a negative impact on the security of software [19]. To tackle the problem Wang *et al.* propose a compiler-based approach to identify pieces of code whose optimised generated code exploit undefined behaviours [20]. We adopt a semantics-based approach that aims at giving a meaning to programs that do not have a defined behaviour according to the ISO C standard.

Memory models have been proposed to ease the reasoning about low-level code. The VCC system [4] generates verification conditions using an abstract typed memory model [5] where the memory is a mapping from typed pointers ($p \in \mathtt{T} \times \mathbb{B}^{|\mathtt{u64}|}$) to structured C values. This memory model is not formally verified. Using the Isabelle/HOL proof assistant, the Autocorres tool [8,9] constructs provably correct abstractions of C programs. Following Tuch *et al.* [18], a concrete memory is abstracted by an abstract memory $m \in 'a\ ptr \rightarrow 'a\ option$ where $'a$ represents the type of the pointer. The memory models of VCC [5] and Autocorres [9] ensure separation properties of pointers for high-level code and are complete with respet to the concrete memory model. For the CompCert memory model [15], separation properties of pointers are for free because pointers are modelled as abstract locations $l \in block \times offset$. For our symbolic extension, the completeness (and correctness) of the normalisation is defined with respect to a concrete memory model and therefore allows to reason about low-level idioms.

Several formal semantics of C are defined over a block based memory model where pointers are modelled by a location $l \in \mathit{block} \times \mathit{offset}$ [7,12,14]. The different models differ upon their precise interpretation of the ISO C standard. The CompCert C semantics [3] provides the specification for the correctness of the CompCert compiler [14]. CompCert is used to compile safety critical embedded systems [1] and the semantics departs from the ISO C standard to capture existing practices. Our semantics extends the existing CompCert semantics and benefits from its infrastructure.

Krebbers also extends the CompCert semantics but aims at being as close as possible to the C standard and proposes a formalisation of sequence points in non-deterministic programs [12] and of strict aliasing restrictions in `union` types of C11 [11]. These aspects are orthogonal to the focus of our semantics which gives a meaning to implementation defined low-level pointer arithmetic. Ellison and Roşu [7] propose an executable C semantics using the K framework [16]. Unlike our semantics with symbolic values, they do not model low-level pointer arithmetic and only have a partial symbolic support for uninitialised values [7, Section 6.2.2].

8 Conclusion

We propose an executable semantics for C programs that augments the block based memory model of CompCert with the ability to reason about low-level pointer arithmetic and uninitialised values. The key insight is the use of symbolic values that represent delayed computations: symbolic values are only normalised when a concrete value is really needed. The normalisation is executable and efficient in practice thanks to the use of SMT solvers.

As future work, we shall investigate how to adapt the correctness proof of the CompCert compiler to our new memory model. A difficulty is that our model makes explicit that the memory is finite as the normalisation exploits the fact that pointers are indistinguishable from C integers. Moreover, our memory model is general enough and should be helpful to add in CompCert new target architectures where integer and float values are not so clearly separated in memory or in registers (e.g. SIMD architecture).

As another line of research, we intend to study how to ground security analyses upon our enhanced memory model. A feature of our memory model is that the normalisation, seen as an SMT query, implicitly enumerates all the possible concrete memory configurations. We shall investigate how to augment the axiomatisation of the memory to assess the consequences of a memory violation (e.g. use-after-free), and perform detailed vulnerability analyses.

References

1. R. Bedin França, S. Blazy, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS2 2012: Embedded Real Time Software and Systems*, 2012.

2. D. J. Bernstein, T. Lange, and P. Schwabe. The Security Impact of a New Cryptographic Library. In *LATINCRYPT'12*, volume 7533 of *LNCS*, pages 159–176. Springer, 2012.
3. S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reasoning*, 43(3):263–288, 2009.
4. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
5. E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A Precise Yet Efficient Memory Model For C. *ENTCS*, 254:85–103, 2009.
6. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
7. C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *POPL*, pages 533–544. ACM, 2012.
8. D. Greenaway, J. Andronick, and G. Klein. Bridging the Gap: Automatic Verified Abstraction of C. In *ITP*, volume 7406 of *LNCS*, pages 99–115. Springer, 2012.
9. D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: formal verification of C code without the pain. In *PLDI*. ACM, 2014.
10. ISO. ISO C Standard 1999. Technical report, 1999.
11. R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*, pages 50–65. Springer, 2013.
12. R. Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in C. In *POPL*, pages 101–112. ACM, 2014.
13. D. Lee. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.
14. X. Leroy. Formal verification of a realistic compiler. *Comm. ACM*, 52(7):107–115, 2009.
15. X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model. In *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
16. D. Lucanu, T. F. Şerbănuţă, and G. Roşu. K Framework Distilled. In *Workshop on Rewriting Logic and its Applications*, volume 7571 of *LNCS*, pages 31–53, 2012.
17. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
18. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108. ACM, 2007.
19. X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? In *APSYS '12*, pages 1–7, 2012.
20. X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *SOSP'13*, pages 260–275. ACM, 2013.