A semantics of structures, unions, and underspecified terms for formal specification

Louis Gauthier, Virgile Prevosto, Julien Signoles

A Semantics of Structures, Unions, and Underspecified Terms for Formal Specification

Louis Gauthier
Virgile Prevosto
Julien Signoles
firstname.lastname@cea.fr
Université Paris Saclay, CEA, LIST
France

ABSTRACT

ACSL is a behavioral interface specification language for C. It is used by Frama-C, a framework including several formal methods-based techniques for verifying C code with respect to ACSL annotations. Currently, there is no formal definition of the ACSL semantics, which may lead to different, possibly inconsistent, interpretations of the semantics by developers and users. This paper is a first step to solve this issue by formalizing a subset of the ACSL specification language in Coq. This semantics is based on Krebbers' semantics of C. The paper focuses on two features: an equality for structures and unions, which are comparable in ACSL, contrary to C, and a logic for handling underspecified terms and predicates that the total logic of ACSL let us manipulate. Finally, we also provide a few properties of our formal semantics.

CCS CONCEPTS

• **Software and its engineering** → **Semantics**; *Formal methods*; **Specification languages**; • **Theory of computation** → **Program semantics**.

KEYWORDS

Formal Semantics, Formal Specification Language, Coq Proof Assistant

1 INTRODUCTION

Software-oriented formal methods offer various techniques for mathematical reasoning on software. When applied on a program written in a specific programming language, they require a formal semantics for this language. For mainstream programming languages such as C, defining such a formal semantics usually requires a tremendous amount of error-prone work. To reduce the risk of introducing errors, it is welcome to define such a semantics in a formal setting like a proof assistant, typically Coq. Once the semantics has been fully defined, it becomes possible to prove some soundness properties over it, such as Milner's famous statement "well-typed programs cannot go wrong" [32]. In addition, it also allows the users to prove the correctness of formal analysis techniques with respect to this semantics of the language, as long as those techniques are formally defined within the same setting.

Several works already define formal semantics of programming languages. Among others, the K framework [25] provides a rewrite-based executable semantic framework in which many semantics have been defined, e.g., for C [14], Java [8], or JavaScript [36]. Similarly, the Coq proof assistant has been used for C [24], Clight [6], which is a large subset of C, or JavaScript [7]. However, fewer works have been dedicated to define the formal semantics of *specification* languages. Such languages are required by many formal method-based techniques, such as deductive verification [16] or runtime annotation checking [22], often referred to as runtime assertion checking [13]. Yet, a few works exist, such as a formalization in Coq of the semantics of JML [28], which is a formal specification language for Java [26] or, more recently, a formal verification in Isabelle of a monitoring framework for Metric First Order Temporal Logic (MFOTL) [40].

This paper contributes to this line of work by proposing a formal semantics written in Coq for a subset of the ACSL specification language [2]. Our long-term goal is to provide a formal semantics for the whole language. Like JML for Java, ACSL is a Behavioral Interface Specification Language (BISL) [17] for C. Notably, it is used by Frama-C [1], a framework for analyzing source code written in C, which provides several techniques for verifying C code with respect to ACSL annotations, such as deductive verification, abstract interpretation, and runtime annotation checking.

Providing a formal semantics of ACSL would help to better define and understand this language. Indeed, its reference manual [2] is not always completely clear about the (informal) semantics of every construct. This may lead to different, possibly inconsistent, interpretations when developing tools or writing specifications. For example, some issues from the Frama-C github show the need of clarification of constructs like the `\valid` [1] or `\initialized` [2] predicates, which are supposed to denote the validity (resp. initialization) of a pointer, but whose definitions in the ACSL manual contain some ambiguities. This formalization would also help validate the different formal method-based techniques used in Frama-C. Actually, several works have already presented correctness proofs of such Frama-C techniques for different subset of C and/or ACSL, but all of them made different choices when simplifying or even modifying the languages' semantics in different incompatible ways according to their needs. Namely, a simple arithmetic ACSL subset for Clight has been introduced in [19] for certified deductive verification. Similarly, a study about certified program slicing is focusing on a While language extended with Boolean assertions [27]. This is

FormaliSE '24, April 14–15, 2024, Lisbon, Portugal
2024. ACM ISBN 979-8-4007-0589-2/24/04…$15.00
https://doi.org/10.1145/3644033.3644380

[1] https://github.com/acsl-language/acsl/issues/74
[2] https://github.com/acsl-language/acsl/issues/81

also the case for a formalization of a method for verifying relational properties [4]. Runtime annotation checking has been studied in [3] on an arithmetic ACSL subset for simple C programs, and on an ACSL subset restricted to some memory properties for a C subset including pointers and addresses in [30].

This paper concentrates on a propositional logic, based on C expressions as terms, including structures and unions, as well as undefined expressions, such as `1/0` and `1/0 == (2-1)/(1-1)`. These constructs are of particular interest. First, ACSL allows the users to compare structures or unions, while it is not possible in C. Such a comparison is in particular not obvious for unions, which are complicated semantic objects. As far as we know, this problem has never been studied (in particular, the ACSL manual does not say much beyond the fact that structs and unions can be "tested for equality" [2, Sect. 2.2.7]). Second, ACSL's logic is total [2, Sect. 2.2.2], meaning that all terms and predicates are well defined. Therefore, the ACSL counterparts of the above-mentioned C undefined expressions must have a well defined, yet underspecified, semantics. Since ACSL relies on C, we based our ACSL formal semantics on Krebbers' formal semantics of C [24], written in Coq. The paper also includes the proof of a few properties of our subset of the ACSL language. To sum-up, our contributions constitute **a first step towards a formal semantics of ACSL, written in Coq**. They include a novel formal semantics of **comparisons over structures and unions**, as well a formal semantics for **underspecified terms and predicates**. In addition, they also provide a proof for **a few properties of this semantics**.

Related Works. As said earlier, several works propose formal semantics of C: CompCert [6] has already been mentioned, but M. Norrish [34], C. Ellison [14], and R. Krebbers [24] have also proposed formal semantics of C respectively formalized in HOL, K, and Coq. But as in C it is not allowed to compare structs or unions, none of these works deal with this subject, which is one of the main focus of our paper. Some works are focusing on C deductive verification like AutoCorres [15] which is translating C code into Isabelle before abstracting the low level representation. It is based on the memory model defined by H. Tuch [39]. This work doesn't handle unions, which is one of our main focus in this paper so we could not rely on this work. Others works consider only specific parts of C. For instance, L. Li [29] introduces a formal model of checked C, a language extending C with new pointer types and annotations, to enforce the safety of memory. Similarly, K. Memarian [31] explores a source-language semantics for memory object and pointers, while D. Chisnall [12] focuses on the semantics of pointer casts. On another topic, K. Nienhuis [33] proposes a formal semantics of C for reasoning on concurrency. Similarly to the works cited above, these formalizations neither introduce structs and unions comparison nor manipulate underspecified terms as we need for ACSL.

Still, some works focus on underspecified or undefined behavior of C. In particular, M. Norrish [35] has proposed a semantics with constraints permitting to C expressions to have a deterministic behavior even in the presence of under-specified behavior of the code with respect to the C standard. Also, C. Hathhorn [18] has introduced a formal semantics that rejects undefined behavior of C. However, in ACSL underspecified terms can be manipulated and compared for example, which is something that none of the works cited earlier deal with. Some formalizations for languages with dependent types or refinement types (refinement type is a type linked to a predicate that must hold for any element of this type) also exist, like MetaCoq [38], which has a Coq formalization, and LiquidHaskell [9] which defines a metatheory based on refinement calculus. These two languages are at a much higher level than C, which implies very different problems. In particular They do not address memory layout issues that are overwhelming in a C formalization. Regarding the formalization of specification languages such as ACSL, as mentioned above, H. Lehner [28] proposes a formalization of JML. As Java does not have the same constructs as C, this work does not deal with structs or unions comparison. P. Chalin [11] presents a sound assertion semantics for the Verifying Compiler project [20], which detects errors such as undefined behavior but, in contrary to what we want, does not allow manipulating underspecified terms. Finally, some works have also been done on the formalization on a subset of ACSL itself, in particular P. Herms [19] or D. Ly [30]. The main focus of their work was not to formalize ACSL, so they chose a subset of ACSL containing only the elements they needed. Hence, structs and unions comparison were not part of their formalization.

Outline. Section 2 introduces a small example of ACSL annotations in a C program, with structs, union, and underspecified terms. Section 3 presents the CH_2O^- language, which is a subset of the CH_2O language introduced by R. Krebbers [24]. Sections 4 and 5 respectively explains the syntax and the semantics of miniACSL, the subset of ACSL we focus on in this paper. Last, section 6 concludes and discusses future works.

2 MOTIVATING EXAMPLE

Fig. 1 presents a C program using a type `simple_packet`. It introduces the most important notions of our work on structures, unions and underspecified terms. A simple packet can either be defined by its source, its destination, its identifier, the length of its content and its content, or simply by a full packet containing all these pieces of information.

The code contains three assertions at lines 32–39, expressed in the ACSL specification language [2] and enclosed in special comments `/*@...*/`.

The first assertion, on line 32, checks that the packets `p1.packet` and `p2.packet` are not equal. These packets are structures. As mentioned in the previous section, such an equality is not allowed in C, so no existing formal semantics of C deals with this construct. However, comparing structures is possible in ACSL. Its reference manual states that *"equality amounts to recursively checking equality of fields* [2, Sec. 2.2.7], so our formalized semantics of ACSL for comparing structures presented in Section 5.1 is a field-based comparison: two structures are equal if and only if all their fields are equal. Accordingly, the assertion at line 32 is valid since the two packets have no fields with the same value.

The second assertion at Line 33 compares two unions `p1` and `p3`. As a reminder, unions in C are field containers but, unlike structures, all fields of a union share the same memory space, as presented in Fig.2. Therefore, changing the value of a field of a given union, change the value of all fields of this union accordingly. Similarly

```
1  struct pack {
2    char src;
3    char dest;
4    char packet_id;
5    char length;
6    unsigned char content[255];
7  };
8  union simple_packet {
9    struct pack packet;
10   unsigned char full_packet[259];
11 };
12
13 int main(){
14   int nb_pack = 0;
15   union simple_packet p1 = {.packet ={
16       .src = 1,
17       .dest = 2,
18       .packet_id = 0,
19       .length = 5,
20       .content = {1,2,3,1,4}}};
21
22   union simple_packet p2 = {.packet ={
23       .src = 2,
24       .dest = 1,
25       .packet_id = 1,
26       .length = 2,
27       .content = {6,9}}};
28
29   union simple_packet p3 = {
30       .full_packet = {1,2,0,5,1,2,3,1,4}};
31
32   /*@ assert(p1.packet != p2.packet);*/
33   /*@ assert(p1 == p3);*/
34   /*@ assert
35      ((p1.packet.length
36        + p2.packet.length
37        + p3.packet.length)
38       / nb_pack)
39      == (5+5+2)/0; */
40   return 0;
41 }
```

Figure 1: Code Using Structures, Unions and Division by Zero.

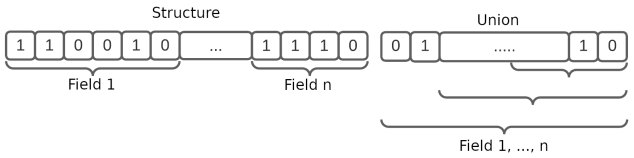

Figure 2: Memory Representation of Structures and Unions.

to structures, unions cannot be directly compared in C, while it is possible in ACSL. In addition to the above quote, the ACSL reference manual warns that *"Beware that equality of unions is also equality of all fields* [2, Sec. 2.2.7]. However, this informal semantics is not precise enough. Typically, in our example, the initialized field of `p1` is `packet`, which is a structure, while the initialized field of `p3` is `full_packet`, which is an array. Since the representations of both types differ, both fields cannot be easily compared each other. An idea could be to check the type of the "active" field of an union (the one with which we initialized the union) in addition to its value, but it does not work well in presence of casts. As detailed in Sec. 5.2, our solution consists in looking at the bit-level representation of unions when comparing them. Here, these representations are the same, so unions `p1` and `p3` are equal.

The last assertion at lines 34–39 compares *underspecified* terms, such as `12/0`. Such terms would lead to an undefined behavior if evaluated as C expressions. However, the ACSL logic is "a 2-valued logic with only total functions. Consequently, logic terms are never "undefined"" [2, Sec. 2.2.2], even if we do not know their values beyond the fact that they belong to their expected types (integers, for terms such as `12/0`). Therefore, it must be possible to use them in ACSL predicates without making the underlying logic inconsistent. Typically, the ACSL equality == must still define an equivalence (i.e., reflexive, transitive, and associative) relation, even when comparing such terms. Here, since the sum of the packets' lengths is equal to 12, the assertion must be valid: if it were invalid, the equality would not be reflexive.

It is worth noting that none of these assertions can be executed in C. Indeed, the first and second assertions compare structures and unions, which is not possible in C, while executing the third one would lead to undefined behaviors when dividing by zero. However, they are legal and well-defined in ACSL. The main purpose of this paper consists in defining a formal semantics for such assertions.

While the code in Fig. 1 presents the main elements of our formalization, it also contains C constructions that are not strictly relevant to the issue at hand. Hence, Fig. 3 contains a simpler example that we will use throughout the rest of this paper. This small

```
1  struct p2D {int x; int y;};
2  struct tri { struct p2D a; struct p2D b; struct p2D c; };
3  struct quad {
4    struct p2D a; struct p2D b; struct p2D c; struct p2D d;
5  };
6  union shape { struct tri tri; struct quad quad; };
7  int main() {
8    struct p2D p1 = {.x=2, .y=1}, p2, p3;
9    p2.x = 2; p2.y = 5;
10   p3.x = 5; p3.y = 1;
11   union shape s1 = {.tri = {.a=p1, .b=p2, .c=p3} };
12   union shape s2 = {.quad = {.a = p1, .b=p2,
         .c=p3,.d={.x=5, .y=8}}};
13   /*@ assert p1 != p2 && p2 != p3 && p3 != p1; */
14   /*@ assert s1 != s2; */
15   return 0;
16 }
```

Figure 3: Example of ACSL Assertions Over structs and unions .

example presents a type `shape` that uses structures and unions. Indeed, a (geometrical) shape is either a triangle or a quadrilateral, both being defined by 2D-points on a plane. The `main` function then defines three points `p1`, `p2`, and `p3`, a triangular shape `s1` and a quadrilateral shape `s2` and contains two ACSL assertions on the equality of the points and on the equality of the shapes.

3 THE CH_2O^- LANGUAGE

As ACSL is meant to formally specify properties over C programs, in order to define the semantics of a subset of ACSL, we first need to

$\circledcirc_c ::= ==|<=|<\quad \circledcirc_a ::= +|-|*|/|\%\quad \circledcirc ::= \circledcirc_c|\ \circledcirc_a\quad \circledcirc_u ::= -|!$
$\alpha ::= := |\ \circledcirc_a := |\ := \circledcirc_a$

e	$::=$	v	constant
	$\mid$	x	variable
	$\mid$	$e._l id$	field access of a left-value
	$\mid$	$e._r id$	field access of a right-value
	$\mid$	$*e$	dereference
	$\mid$	$\&e$	address
	$\mid$	load e	loading from memory
	$\mid$	$(\tau)e$	cast operator
	$\mid$	$\circledcirc_u e$	unary operation
	$\mid$	$e_1 \circledcirc e_2$	binary operation
	$\mid$	$e_1 \alpha e_2$	assignments
	$\mid$	$e_1[\vec{r} := e_2]$	altering structs and unions
s	$::=$	e	expression
	$\mid$	skip	skip
	$\mid$	return e	return
	$\mid$	if (e) s_1 else s_2	conditional
	$\mid$	$s_1; s_2$	sequence
	$\mid$	while(e) s	loop
	$\mid$	local$_\tau s$	local variable declaration

Figure 4: CH_2O's Syntax.

have a semantics for the corresponding subset of C. As mentioned above, several C formal semantics have already been defined, so there was no need to define our own. In particular, there are the formal C semantics of C. Ellison and G. Rosu [14], the one used in Cerberus by K. Nienhuis, K. Memarian and P. Sewell [33], the one of M. Norrish in HOL [34], and the one used by the CompCert certified compiler [6] in Coq. All these semantics were not chosen because their representations of values did not suit our needs. Typically, our semantics of unions in ACSL, presented in Sec. 5, requires a bit-level representations of values that these semantics do not provide. Hence, we have chosen to rely on Krebbers' CH_2O [24], written in Coq. More precisely, we extract CH_2O^-, a subset of CH_2O that is large enough for our study. Note that we only focus on CH_2O^- in this paper in order to have a clear self-contained presentation of the language. However, our Coq development [3] is based on the whole CH_2O development.

The rest of this section presents CH_2O^-. Its content is *not* new, and reuses Krebbers' notations whenever possible. However, this section is necessary to introduce the grounding blocks required by our semantics of miniACSL. Sec. 3.1 introduces the CH_2O^-' syntax and type system, while Sec. 3.2 presents its memory model.

3.1 CH_2O^-'s Syntax and Type System

Fig. 4 describes CH_2O^- syntax, composed of C expressions and statements. Expressions include integer constants, variables and field accesses, unary and binary operations on integers and pointers, pointer dereferences, addresses of an expression and casts of an expression to a given type, assignments, loading a value from memory and altering structs and unions.

[3] https://github.com/Anonym398/minACSL

$k \in K ::= \text{char} \mid \text{short} \mid \text{int} \mid \text{long} \mid \text{long long} \mid \text{ptr\_rank}$
$si ::= \text{signed} \mid \text{unsigned}\quad \tau_i ::= si\ k\quad \tau_p ::= \tau\quad \tau_b ::= \tau_i \mid \tau_p*$
$\tau \in \text{type} ::= \tau_b \mid \text{struct } t \mid \text{union } t$

Figure 5: CH_2O^-'s Types.

In CH_2O^-, left-values are not converted to right-values implicitely as it is done in C by *lvalue conversion* [23]. load expressions materialize reading a value from memory. The argument of load must be a *left-value*, in C sense, i.e., a location in memory. Then, load retrieves the value stored in this location. Similarly, field accesses are tagged to indicate whether they are used as left- or right-values (respectively $e._l id$ and $e._r id$).

Binary operations contains comparison and arithmetic operations. It is worth noting that the comparison operators can only be applied on integers and pointers: In C, and so in CH_2O^-, it is not possible to compare structures or unions. We then find assignment expressions, which include simple assignment (:=), compound prefix operations ($\circledcirc_a :=$) and postfix increment and decrement ($:= \circledcirc_a$).

Finally, in CH_2O^-, _[_ := _] is used to transcribe C's *compound literals*. More precisely, such an expression denotes a compound literal in which a field has been given a value. Then, a compound literal is translated as a sequence of such insertions, starting with an empty compound noted 0^τ.

For instance, the CH_2O^- expression corresponding to the initialization of p1 on line 8 of Fig. 3 is:

$$0^{\text{struct } p2D}[(\xleftarrow{\text{struct } p2D} 0) := \text{int}_{int}\ 2][(\xleftarrow{\text{struct } p2D} 1) := \text{int}_{int}\ 1].$$

In this example, $0^{\text{struct } p2D}$ is the empty compound literal of type $p2D$. First, $[(\xleftarrow{\text{struct } p2D} 0) := \text{int}_{int}\ 2]$ corresponds to the assignment .x = 2, with $(\xleftarrow{\text{struct } p2D} 0)$ corresponding to the field index (x is the first field), and $\text{int}_{int}\ 2$ being its value. Similarly, the second insertion corresponds to the initialization of .y to the value 1.

Statements include expression evaluations, skip (i.e., the effectless statement), return, conditionals, sequences and while loops. There is also a special statement used for local variable declaration. Indeed the local$_\tau s$ statement opens a block scope with a local variable of type τ.

For typing CH_2O^-, we directly use the C type system defined by Krebbers [24], restricted to the types of CH_2O^-. Our types are shown in Fig. 5. The set K contains standard C's ranks of integers, as well as a specific rank corresponding to C's intptr_t, to represent the numerical values of pointers. As usual in C, signedness will often be omitted, and considered signed by default. Unions and structures types are defined by their names $t \in \text{tag}$ represented as strings.

Let index be a countable set and $\mathbb{B}$ be the set of Booleans. Each object has an object identifier $o \in \text{index}$. A memory typing environment $\Delta \in \text{memenv} := \text{index} \to \text{type} \times \mathbb{B}$ is a partial function mapping objects to their type and danglingness status.

We do not specify here Krebbers' typing rules restricted to the CH_2O^- subset. However, we assume that programs are well typed according to these rules, which means in particular:

- Each expression e has a type τ in an adequate memory typing environment Δ, which is noted $\Delta \vdash e : \tau$.

- Each object identifier corresponding to a program object has a type τ, with a danglingness status $b \in \mathbb{B}$, in an adequate memory typing environment Δ, which is noted $\Delta(o) = (\tau, b)$
- An object is said *alive* in a memory typing environment Δ, which is noted $\Delta \vdash o$ alive, when $\Delta(o) = (\tau, \text{false})$.
- A tag environment $\Gamma \in \text{env}$ assigns fields to structs and unions, i.e., env := tag $\rightarrow$ list type. It basically links a tag corresponding to a struct or union to the list of types, each of them corresponding to a field of the struct or union.

3.2 Memory Objects and Values

In this paper, CH_2O^- objects that are not integers are named *memory objects*: their semantics rely on their memory representation, which is presented here.

The part of an object that is accessed through a field i of a structure or an union o is a *reference*. It is represented by a path from the top of o to i. Such a path is a list of *reference segments*, corresponding to field accesses. The empty reference ϵ contains no segment. References and their segments are defined as follows:

$$r \in \text{refseg} ::= \xrightarrow{\text{struct } t} i \mid \xrightarrow{\text{union } t} i \qquad \text{Reference segment}$$
$$\vec{r} \in \text{ref} := \text{list refseg} \qquad \text{Reference}$$

Addresses are represented by an associated object identifier o of type τ, a reference $\vec{r}$ to some sub-object of type σ, and a type σ_p to represent a cast. A pointer is either an address or the NULL pointer of type σ_p. More precisely, addresses and pointers are defined as follows:

$$a \in \text{addr} ::= (o : \tau, \vec{r})_{\sigma > \star \sigma_p} \qquad \text{Address}$$
$$p \in \text{ptr} ::= \text{NULL}_{\sigma_p} \mid a \qquad \text{Pointer}$$

For any address a and reference segment r, $a\langle r \rangle_\Gamma$ is the address of the field accessed through r in the object at a. It is defined by:

$$a\langle r \rangle_\Gamma = (o : \tau, \vec{r}, r)_{\sigma_2 > \star \sigma_2} \text{with } a = (o : \tau, \vec{r})_{\sigma > \star \sigma_p}$$

Here, σ_2 is the type of the field represented by r in the object at address a. For instance, consider the following declaration.

```
struct S { int *p; union U {char* x; int y ;} u; } s;
```

Assuming the object identifier of `s` is o_s, the representation of the pointer `(int**)&s.u.x` is the following address:

$$(o_s : \text{struct } S, \xrightarrow{\text{struct } S} 1 \xrightarrow{\text{union } U} 0)_{\text{char*} > \star \text{int*}}.$$

Starting from o_s, we take the second field of **`struct`** `S`, then the first field of **`union`** `U`, to obtain the address of a **`char`** `*`, that we convert to the address of a **`int`** `*`.

Bits Representation. The C language allows for bit-level manipulations of objects. Therefore, we need a bit-level representation of their values. We reuse Krebbers' bit representation:

$$b \in \text{bit} ::= 0 \mid 1 \mid (\text{ptr } p)_i \mid \lightning$$

In most cases, a bit is either 0 or 1. $(\text{ptr } p)_i$ denotes the i^{th} bit of a pointer p. $\lightning$ represents an indeterminate bit [23, Sect. 3.19.2].

For example, consider a little-endian architecture where an **`int`** contains 32 bits and a **`short`** contains 16 bits, and a type **`union`** `ex {int i; short s};`. Then, the bit representation of variable `e1` of type **`union`** `ex` initialized by `{.s = 257}` is the following:

$$\overrightarrow{00000001}\ \overrightarrow{00000001}\ \overrightarrow{\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning}\ \overrightarrow{\lightning\lightning\lightning\lightning\lightning\lightning\lightning\lightning}.$$

The last 16 bits are indeterminate because `e1` has been initialized through its short `s` field: these bits have not been initialized, so they have no known values. In this paper, a byte is as usual a vector of 8 bits. Inside a byte, if n consecutive bits have the same value x, we group them into one block of bits noted x_n. For example, the bit representation of `e1` will be simply noted $\overrightarrow{0_7 1}\ \overrightarrow{0_7 1}\ \overrightarrow{\lightning_8}\ \overrightarrow{\lightning_8}$.

Values. A higher-level representations of values is defined in Fig. 6. A base value can be an integer $\text{int}_{\tau_i}\ x$, defined by its type

$$v_b \in \text{baseval} ::= \text{int}_{\tau_i}\ x \mid \text{ptr } p \mid \text{indet } \tau_b$$
$$v \in \text{val} ::= v_b \mid \text{struct}_{tag}\ (\vec{v}) \mid \text{union}_{tag}\ (i, v) \mid \overline{\text{union}_{tag}}\ (\vec{v})$$

Figure 6: CH_2O^-'s Values.

τ_i and its numerical value $x \in \mathbb{Z}$, a pointer p, or an indeterminate value of type τ_b. The value $\text{int}_{\tau_i}\ x$ will be shortened to x when there is no ambiguity. A structure is defined by a sequence of sub-values, corresponding to the values of each of its fields. Unions have two different representations depending on how they have been created. First, field-based representation $\text{union}_t\ (i, v)$ means that its current value has been obtained by assigning v to its i^{th} field. This representation is used to enforce C's *type-punning* constraints. Type-punning consists in accessing a union through another field than the one with which the union was written to. Enforcing type-punning constraints must obey some constraints, so that the union value must remember, whenever possible, this field. Indeed, type-punning does not have the same behaviour in all cases so this representation is needed. This representation is not needed for structures as all fields have their own place in memory and so the notion of type-punning is not applicable there. Second, $\overline{\text{union}_t}\ (\vec{v})$ denotes the extensive representation of an union resulting from some low-level operation that breaks its field-based representation, typically through a byte-by-byte copy. Vector $\vec{v}$ represents the value of its bit representation as interpreted according to the type of each field of the union.

As an example of a union value of the first form, consider the initialization `{.tri={.a=p1,.b=p2,.c=p3}}` of `s1` at line 11 of Fig. 3. In that case, the union is defined by an integer index (here .0 since `.tri` is the first element) and its associated value (here, a **`struct`** `tri` containing `p1`, `p2`, and `p3`). Therefore, its representation is:

$$\text{union}_{shape}\ (.0, \text{struct}_{tri}\ (p1, p2, p3))$$
$$p1 = \text{struct}_{p2D}\ (2, 1)$$
$$p2 = \text{struct}_{p2D}\ (2, 5)$$
$$p3 = \text{struct}_{p2D}\ (5, 1)$$

On the other hand, consider now the code below, which initializes the union `u` one byte after the other through a **`for`**-loop[4]:

[4]This example is written in C, but a similar version can be written in CH_2O^-.

```
1 int main() {
2   union s_o_i { short s; int n; } u;
3   for (size_t i=0; i<sizeof(u)-1; i++)
4     ((unsigned char*) &u) [i] = 0;
5 }
```

In that case, the union has not been properly initialized through one of its field, so we only know its extensive bit representation, which is $\overrightarrow{0_8}\ \overrightarrow{0_8}\ \overrightarrow{0_8}\ \overrightarrow{\lightning_8}$. Indeed, the `for` loop initializes the first three bytes to 0 but not the last one. The first two bytes are the bytes corresponding to the s field of u, so this field is initialized and its value is $\mathsf{int}_{short}\ 0$. However, the field n of u has four significant bytes, so one of them is not initialized. Consequently, the value associated to this field is $\mathsf{indet}_{\mathsf{int}}$. Therefore, u's value is as follows:

$$\overline{\mathsf{union}_{s\_o\_i}}\ (\mathsf{int}_{\mathsf{short}}\ 0, \mathsf{indet}_{\mathsf{int}}).$$

Function flatten : val → list bit converts a higher-level value to its corresponding sequence of bits. The exact definition of this function is omitted here (see [24] for its definition on $\mathsf{CH_2O}$). In the rest of the paper, we will note $\overline{v} = \mathsf{flatten}(v)$ for any value v. This function will notably be useful when we define our notion of equality on unions in Sect. 5.2. For instance, flattening the value of u of the previous example after its initialization results in:

$$\mathsf{flatten}(\mathsf{u}) = \overline{\mathsf{u}} = \overrightarrow{0_8}\ \overrightarrow{0_8}\ \overrightarrow{0_8}\ \overrightarrow{\lightning_8}.$$

Equality of Values. Equality of $\mathsf{CH_2O}^-$ values only deals with integers and pointers. It is defined as follows:

$$\mathsf{int}_{\tau_1}\ x_1 =_v \mathsf{int}_{\tau_2}\ x_2 \text{ iff } \begin{cases} \mathsf{int\_cast}(\tau, x_1) =_{\mathbb{Z}} \mathsf{int\_cast}(\tau, x_2); \text{and} \\ \tau =_\tau \mathsf{int\_promote}(\tau_1) \cup \mathsf{int\_promote}(\tau_2) \end{cases}$$

$$\mathsf{ptr}\ a_1 =_v \mathsf{ptr}\ a_2 \text{ iff } \begin{cases} a_1 =_{addr} (o_1 : \tau_1, \vec{r_1}); \text{and} \\ a_2 =_{addr} (o_2 : \tau_2, \vec{r_2}); \text{and} \\ o_1 =_{obj} o_2; \text{and} \\ oo_1 := \mathsf{addr\_object\_offset}(a_1); \text{and} \\ oo_2 := \mathsf{addr\_object\_offset}(a_2); \text{and} \\ oo_1 =_{offs} oo_2 \end{cases}$$

$\mathsf{ptr}\ \mathsf{NULL}_{\sigma_1} =_v \mathsf{ptr}\ \mathsf{NULL}_{\sigma_2}$

This definition is based on a few operations that are not introduced in this paper for the sake of concision: they are not critical for the understanding of our own contribution. Basically, before being compared, integers are converted into a common type τ by function int_cast, following the promotion rules of the C standard given by function int_promote. Pointers are equivalent when both their corresponding objects and the offsets in these objects, given by function addr_object_offset, are the same. Finally, NULL pointers are equivalent, independently from their type. We assume that this Krebbers' equality defines an equivalence (i.e., transitive, reflexive and symmetric) relation. Proving it is left for future works.

4 miniACSL SYNTAX

This section presents the syntax of our formal specification language, named miniACSL, and shown in Fig. 7. It extends $\mathsf{CH_2O}^-$ statements by introducing logical assertions. Assertions make the link between $\mathsf{CH_2O}^-$ and miniACSL. Informally, they state that a miniACSL predicate must be true at a given program point. Terms are the logical counterpart of $\mathsf{CH_2O}^-$ expressions. More precisely,

$\circledcirc_l$	::=	$\wedge \mid \vee \mid \Rightarrow$	
t	::=	v	constant
		x	variable
		$t.{}_l id$	field access of a left-value
		$t.{}_r id$	field access of a right-value
		$-t$	unary operation
		$t_1 \circledcirc_a t_2$	binary operation
		$*t$	dereference
		$\&t$	address
		load t	loading from memory
		\baseaddress(t)	pointer's base address
		\offset(t)	pointer's offset
		\blocklength(t)	pointer's block length
$\wp$	::=	True | False	propositional constant
		$t_1 \circledcirc_c t_2$	comparison operation
		$\wp_1 \circledcirc_l \wp_2$	logical connector
		$\neg\wp$	negation
		\valid(t)	pointer's validity
		\initialized(t)	pointer's initialization
s	::=	...	existing $\mathsf{CH_2O}^-$ statements
		assert($\wp$)	miniACSL assertion

Figure 7: miniACSL **Syntax.**

terms include "pure" $\mathsf{CH_2O}^-$ expressions, i.e. excluding assignments. In miniACSL, terms also do not include comparison operations, which are lifted to predicates. Similarly, the only unary operator is the unary minus, since the negation operator is lifted to predicates. Our terms include in addition three ACSL's built-in logic functions that are used in specifications about pointers. First, the function \baseaddress(t) returns the base address of a pointer t, i.e. the first address of the memory block containing t. Second, the function \offset(t) returns the offset of a pointer t, i.e. the number of bytes between the address where t points to and its base address. Third, the function \blocklength(t) returns the length in bytes of the memory block where the pointer t points to. Predicates include the two propositional constants True and False, term comparisons, standard logical connectors (conjunction, disjunction, and implication), and negation. They also include two ACSL's built-in predicates related to memory properties: Predicate \valid(t) is valid if and only if the pointer t is valid, meaning that the memory location it points to can be safely written, and predicate \initialized(t) is valid if and only if every non-padding bit of the memory location t points to has been properly initialized with some determinate value.

5 miniACSL SEMANTICS

We use a *shallow embedding* [10] to define our semantics of miniACSL. It means that we rely on Coq's built-in elements in our development, in particular on the following Coq operators:

$$\circledcirc_{lc} \in \mathsf{logicalcoqop} ::= \overset{c}{\wedge} \mid \overset{c}{\vee} \mid \overset{c}{\Rightarrow}$$

$$\circledcirc_{ac} \in \mathsf{arithcoqop} ::= \overset{c}{+} \mid \overset{c}{-} \mid \overset{c}{*} \mid \overset{c}{/}$$

Moreover, contrarily to C, where every integer type has a finite size, ACSL integers are mathematical, unbounded integers. Hence,

miniACSL values, called valacsl, extend CH_2O^- values with plain mathematical integers without any size or signedness information:

$$\begin{array}{rll} v_a \in \text{valacsl} ::= & v & CH_2O^- \text{ value} \\ & \mid \text{int}_z\ x & \text{mathematical integer} \end{array}$$

Sect. 5.1 (resp. 5.2) presents structure (resp. union) comparison. Then, Sect. 5.3 focuses on underspecified terms. Finally, Sect. 5.4 and 5.5 detail the formal rules for evaluating terms, predicates and assertion, respectively.

5.1 Structure Comparison

As already mentioned, ACSL allows comparing two unions or two structures, so that we have to extend $=_v$ to these values. For structures, we just compare pairwise each field of both values. This is similar to structural comparisons that exist in many programming languages, e.g., equality = in OCaml.

The equality between two structures is defined as follows:

$$\forall t_1, t_2, \vec{v_1}, \vec{v_2},\ \text{struct}_{t_1}(\vec{v_1}) =_v \text{struct}_{t_2}(\vec{v_2}) \leftrightarrow \forall i, v_{1_i} =_v v_{2_i}$$

Consider the comparison `p1 != p2` at line 13 of Fig. 3. The values of `p1` (resp. `p2`) is $\text{struct}_{p2D}\ (\text{int}_{int}\ 2, \text{int}_{int}\ 1$ (resp. 5)) at that program point. Comparing these structures means comparing each of their fields, pairwise. Hence, `p1 != p2` is evaluated to True because the values for the second field of both structures are not the same.

Note that we do not take tags into account for comparing structures, so two structures with differents tags can be equal, provided they have the same number of fields (each of them having compatible types). For example, let us assume we have another structure type: `struct sh {int a; short b;};` and a structure `s` represented by $\text{struct}_{sh}\ (\text{int}_{int}\ 2, \text{int}_{short}\ 1)$. Here, `s` and `p1` are equal according to our comparison, even if they do not have the same tags for their second field. Such structures can indeed be compared since types `int` and `short` are compatible. It allows comparing the second field of each structure. Notice that, in contrary to the comparison of unions explained in the following section, comparison of structures does not need the low level representation introduced in Section 3.2 as structures have only one representation in our formalism which means our field base equality handle all possible structure comparisons.

5.2 Union Comparison

Contrary to structures, unions have several representations depending on how we initialized it (byte by byte, or using one of its field), as already explained in Section 3.2. Hence, comparing unions is trickier than comparing structures. A first possibility is to differentiate both cases. Indeed, in case both unions have a field-based representation, we could try to only compare their used field and associated value. Conversely, in case both are an extensive representation, we compare all values of their list of values (similarly to what we do with structures). Finally, if one union has a field-based representation and the other one an extensive representation, it could seem natural to pick up the value in the vector of the latter corresponding to the field of the former. However, such a relation would *not* be transitive. Let us take as example the three unions `u1`, `u2` and `u3` defined in Fig. 8. Assuming a little-endian architecture with

```
1  int main() {
2    union c_o_i { char c; int n; } u1, u2, u3;
3    u1.c = 0;
4    for (size_t i = 0; i = sizeof(u2); i++)
5      ((uint8_t*) &u2) [i] = 0;
6    ((uint8_t*) &u2) [1] = 1;
7    u3.n = 256;
8  }
```

Figure 8: Code Initializing Unions u1, u2, and u3.

32-bit `int` the values of u1, u2, and u3 are $\text{union}_{c\_o\_i}\ (.0, \text{char}\ 0)$, $\overline{\text{union}_{c\_o\_i}}\ (\text{char}\ 0, \text{int}\ 256)$, and $\text{union}_{c\_o\_i}\ (.1, \text{int}\ 256)$, respectively. As `u1` is properly initialized by its `c` field (which is the first field of the `union c_o_i` type), we can represent the union with its field representation using .0 to specify that the field initialized is the first one and using char 0 to indicate the value of this field. The same process can be applied to `u3` which is also initialized by one of its field. The value of `u2` is based on its extensive representation. Indeed, we initialize `u2` byte by byte, so its representation in memory is $\overrightarrow{0_8}\ \overrightarrow{10_7}\ \overrightarrow{0_8}\ \overrightarrow{0_8}$ since we initialized all its bytes to 0 but the second, set to 1. Hence, the `char` field of `u2`, corresponding to the first byte, has a value of 0. Similarly, its `int` field extends on all four bytes, corresponding to a value of 256. Since we know that `u1` has been set through its `.c` field, we could compare their values u_1 and u_2 by observing only their field 0. Therefore, since $u_1.0 = 0$ and $u_2.0 = 0$, $u_1 = u_2$. Similarly, u_3 is set through its `.n` field, at index 1, and we have $u_2.1 = 256$ and $u_3.1 = 256$. Therefore, we would also have $u_2 = u_3$. However, $u_1 = u_3$ does not hold with our definition, since u_1 and u_3 are defined with distinct fields.

After exploring several possibilities, we eventually chose to use a bit-by-bit comparison in all cases. Indeed, all others explored solutions were in fact not equivalence relations. More precisely, our bit-by-bit comparison relies on the flatten function of Sec. 3.2 to get the binary representations of the compared unions:

$$\forall u_1, u_2 \in \{\text{union}; \overline{\text{union}}\}, u_1 =_v u_2 \leftrightarrow \overline{u_1} \doteq \overline{u_2}$$

Equality $\doteq$ of two binary representations is defined as follows:

$$\begin{array}{l} \forall b_1, b_2 \in \text{bit}, b_1 \doteq b_2 \leftrightarrow \\ \quad (b_1 = n \wedge b_2 = n \wedge n \in \{0; 1; \lightning\}) \\ \vee\ (b_1 = (\text{ptr}\ p_1)_{i_1} \wedge b_2 = (\text{ptr}\ p_2)_{i_2} \wedge \text{ptr}\ p_1 =_v \text{ptr}\ p_2 \wedge i_1 = i_2) \end{array}$$

Theorem 5.1 (The relation $=_v$ is an equivalence relation). *The relation $=_v$ is reflexive, symmetric, and transitive.*

Sketch of proof. The proof is part of our Coq development[5]. This relation on bits is defined so as to ensure reflexivity: it is defined by induction on the structure of bits and only relies on equivalence relation between subterms. In particular, two indeterminate bits are always equal, but different from the other bits. □

Going back to the example of Fig.8, we would have the representations $\overrightarrow{0_8}\overrightarrow{\lightning_8}\overrightarrow{\lightning_8}\overrightarrow{\lightning_8}$, $\overrightarrow{0_8}\overrightarrow{10_7}\overrightarrow{0_8}\overrightarrow{0_8}$, and $\overrightarrow{0_8}\overrightarrow{10_7}\overrightarrow{0_8}\overrightarrow{0_8}$ for $\overline{u1}$, $\overline{u2}$, and $\overline{u3}$, respectively. With our new equality, we see that $\overline{u2} \doteq \overline{u3}$ so u2 and u3 are considered equals with our equality while u1 and u2 are not.

[5] https://github.com/Anonym398/minACSL

5.3 Underspecified Terms

This section presents our semantics for underspecified terms and particularly for terms resulting in a division by zero. Indeed, as mentioned in Section 2, ACSL allows manipulating underspecified terms. For example, by reflexivity of equality, we should be able to prove that `4/0 == 4/0` holds.

For solving this issue, we introduce a new function $\mathsf{divZ} : Z \to Z$ denoting a division by zero. Its parameter represents the numerator of the division. This function is left fully unspecified: we only know that its parameter is any integer and it returns any integer. Yet, it is enough to prove for instance that, for any integer x, $\mathsf{divZ}(x)$ is equal to itself. Thanks to this constructor, we can define the semantics of the miniACSL arithmetic operations $op_z \in \circledcirc_z := \{/_z; \times_z; +_z; -_z\}$ as follows:

$$\mathsf{int}_z\ x\ op_z\ \mathsf{int}_z\ y := \begin{cases} \mathsf{int}_z\ x \overset{c}{/} y & \text{if } y \neq 0 \text{ and } op_z = / \\ \mathsf{int}_z\ \mathsf{divZ}(x) & \text{if } y = 0 \text{ and } op_z = / \\ \mathsf{int}_z\ x \overset{c}{op}\ y & \text{if } op_z \in \{+_Z; -_Z; \times_Z\} \end{cases}$$

Thanks to this definition, the term `4/0` can be evaluated to $\mathsf{divZ}(4)$ and manipulated in Coq like any other integer. In particular, it allows us to use the existing Coq logic and tactics on Z in our meta-theory. Let us illustrate this with the predicate `5+4/0-2 == 1+(x+2)/0+2`, assuming the value of `x` of type **`int`** is 2. This assertion is true. Thanks to our Coq formalisation, this is very easy to prove. Indeed, the terms $1 + \mathsf{divZ}(x + 2) + 2$ can be rewritten to $1 + \mathsf{divZ}(4) + 2$ since x is equal to 2 and, by using standard properties over integers, the terms $5 + \mathsf{divZ}(4) - 2$ and $1 + \mathsf{divZ}(4) + 2$ can be rewritten to the same term, for instance $3 + \mathsf{divZ}(4)$, which is equal to itself by reflexivity. The Coq's proof is part of our coq development, in `core_c/proof_paper.v`.

Division by zero is not the only underspecified term existing in ACSL. In particular, any invalid memory access, such as dereferencing an invalid pointer, is also underspecified. It could be handled similarly by introducing a new underspecified function for each kind of error. For instance, the semantics of `*p` when the value of `p` is `NULL` would be $\mathsf{deref}(\mathsf{NULL})$ with deref an underspecified function. This extension is left to future work.

Our work is slightly different from existing works on undefined behavior. Indeed, C. Hathhorn [18] defines a semantics which rejects undefined behavior in C by for example, adding restrictions to semantic rules in a way of never having to deal with an undefined behavior. Similarly, M. Norrish [35] represents undefinedness in C by a unique value, which is the same for every undefined expression. P. Herms [19] uses the mechansims of opaque definitions and subset types of Coq to handle underspecified terms in ACSL, but without being allowed to manipulate these underspecified terms as we can do with our semantics.

5.4 Semantics of miniACSL Terms

Our semantics for miniACSL terms is a denotational semantics similar to $\mathsf{CH_2O}$'s one for pure expressions. In particular, we reuse some elements (slighly simplified in this paper):

- A stack ρ is a function that takes a variable x as argument and returns its object identifier o and its type τ: $\rho(x) = (o, \tau)$.
- A memory m maps an address a to a value v: $m(a) = v$.
- The memory typing environment associated to a memory m, noted $\overline{m}$, that associates to every address of m the type and the danglingness status of the stored value.
- $\mathsf{size\_of} : \mathsf{type} \to \mathbb{N}$ returns the size of any type in bytes.
- $\mathsf{top} : \mathsf{index} \to \mathsf{type} \to \mathsf{addr}$ denotes the top address of an object identifier o, i.e., $\mathsf{top}_\tau(o) = (o : \tau, \epsilon)_{\tau > \star \tau}$.
- $\mathsf{ref\_offset}_\Gamma : \mathsf{addr} \to \mathbb{N}$ denotes the offset of a reference $\vec{r}$.

For example, $\mathsf{ref\_offset} \xhookrightarrow{\text{struct } tri} 1 \xhookrightarrow{\text{struct } p2D} 1 = 12$. Indeed, it accesses the second field of a **`struct`** `p2D` stored in the second field of a **`struct`** `tri`, while the first field of a triangle is 8-byte long and the first field of a `p2D` is 4-byte long (assuming **`int`** is 4-byte long and no padding bytes due to alignment).

We note $[\![t]\!]^{\Gamma,\rho,m}_{\mathcal{T}}$ the evaluation of the term t in our denotational semantics. This function depends on the stack ρ, the memory m and the tag environment Γ. For brevity, we omit ρ, m, Γ and $\mathcal{T}$ that only indicates that the evaluation is the one of the term, and simply use the notation $[\![t]\!]$ since the omitted parameters are provided by the C context and are constant in the logical world. Fig. 9 presents the rules for evaluating a term either to a value $v_a \in \mathsf{valacsl}$ (right value), or an address $a \in \mathsf{addr}$ (a left value).

1.	$[\![v]\!]$	$= v$	
2.	$[\![x]\!]$	$= \mathsf{top}_\tau(o)$	if $\rho(x) = (o, \tau)$
3.	$[\![\mathsf{load}\ t]\!]$	$= m(a)$	if $[\![t]\!] = a$
4.	$[\![t.{}_l r]\!]$	$= a\langle r \rangle$	if $[\![t]\!] = a$
5.	$[\![t.{}_r r]\!]$	$= v[r]_\Gamma$	if $[\![t]\!] = v$
6.	$[\![-t]\!]$	$= -[\![t]\!]$	
7.	$[\![t_1 \circledcirc_a t_2]\!]$	$= [\![t_1]\!]\ op_z\ [\![t_2]\!]$	
8.	$[\![*t]\!]$	$= p$	if $[\![t]\!] = \mathsf{ptr}\ p$
9.	$[\![\&t]\!]$	$= \mathsf{ptr}\ [\![t]\!]$	
10.	$[\![\backslash\mathsf{baseaddress}(t)]\!]$	$= \mathsf{top}_\tau$	if $[\![t]\!] = \mathsf{ptr}\ (o : \tau, \vec{r})$
11.	$[\![\backslash\mathsf{offset}(t)]\!]$	$= \mathsf{int}_Z\ \mathsf{ref\_offset}\ \vec{r}$	if $[\![t]\!] = \mathsf{ptr}\ (o : \tau, \vec{r})$
12.	$[\![\backslash\mathsf{blocklength}(t)]\!]$	$= \mathsf{int}_Z\ \mathsf{size\_of}_\Gamma \tau$	if $[\![t]\!] = \mathsf{ptr}\ (o : \tau, \vec{r})$

Figure 9: Evaluation of Terms.

The evaluation of a value is simply the value itself. Variable evaluation consists in looking up its object identifier o in the stack ρ and returning the corresponding address.

As said above, the load expression gets the value stored at the given address. Case 4 evaluates a field access as an address: the function returns the address of the field accessed. Case 5 is the case of a right-value, so the term is evaluated as a value $v \in \mathsf{valacsl}$: the function returns the value $v[r]_\Gamma$, which is the value of the field access represented by the reference segment r for the value v. The & operation takes an address (left-value) in argument and returns a pointer (right-value) pointing to this address. The $*$ operation takes a pointer value in argument and returns a left-value corresponding to the address that is pointed to by the value in argument.

For arithmetic operations, however, we need to modify the $\mathsf{CH_2O}$ semantics since we use mathematical (i.e. unbounded) integers, so that, contrarily to C, there is no overflow. We rely on our definition $op_z \in \circledcirc_z$ for arithmetic operations that allows for using $\mathsf{divZ}(x)$. For example, $[\![3 + 4/0]\!] = \mathsf{int}_z\ 3 +_z \mathsf{int}_z\ \mathsf{divZ}(4) = \mathsf{int}_z\ 3 \overset{c}{+} \mathsf{divZ}(4)$.

For evaluating a call to one of the three built-in logic functions, we evaluate their argument as a pointer value (right-value), and then we can rely on the functions presented above. Assuming t evaluates to a pointer value ptr p with $p = (o : \tau, \vec{r})_{\sigma > \star \sigma_p}$, \baseaddress($t$) uses top to retrieve o. Similarly, \offset(t) uses the function ref_offset on $\vec{r}$ to get its offset from the pointer p, and \blocklength(t) uses the function size_of on the type τ.

5.5 Semantics of miniACSL Predicates

Our predicates are evaluated to Coq's `Prop` type. This shallow embedding avoids the need to reimplement our whole logic in Coq and allows us to easily use Coq's tactics and standard library in our proofs. In other words, our evaluation function for predicates does not return a truth value but is rather a conversion from a miniACSL predicate to a Coq Prop. Hence, to verify a predicate is True we have to prove its translation into Coq's `Prop`.

Fig. 10 presents the evaluation of predicates. As for terms, we define a denotational semantics noted $[\![\wp]\!]^{\Gamma,\rho,m}_{\mathcal{P}}$ that returns the translation of a predicate into a Coq Prop. As with terms, we will omit the parameters, since they do not change during evaluation. We introduce the judgment $\vDash [\![\wp]\!]$. This judgement means that a proof of this Coq Prop exists in the formalism (so the predicate $\wp$ holds). To convert a relational operation between two terms,

$$
\begin{array}{lll}
1. & [\![True]\!] & = True \\
2. & [\![False]\!] & = False \\
3. & [\![t_1 \circledcirc_c t_2]\!] & = \mathsf{comp\_val}(\circledcirc_c, [\![t_1]\!], [\![t_2]\!]) \\
4. & [\![\wp_1 \circledcirc_l \wp_2]\!] & = [\![\wp_1]\!] \circledcirc_{lc} [\![\wp_2]\!] \\
5. & [\![\neg\wp]\!] & = \overset{c}{\neg} [\![\wp]\!] \\
6. & [\![\backslash\mathsf{valid}(t)]\!] & = \begin{cases} True & \text{if } [\![t]\!] = \text{ptr } a \text{ and } \overline{m} \vdash a \text{ alive} \\ False & \text{if } [\![t]\!] = \text{ptr } a \text{ and } \neg\overline{m} \vdash a \text{ alive} \end{cases} \\
8. & [\![\backslash\mathsf{initialized}(t)]\!] & = \mathsf{init\_val}(v) \quad \text{if } [\![t]\!] = \text{ptr } a \text{ and } m(a) = v
\end{array}
$$

Figure 10: Evaluation of Predicates.

we have defined a Coq inductive definition comp_val. For each logic connector we simply convert the operand(s) and use the corresponding Coq connector. Similarly, we introduce a new Coq inductive definition init_val that allows us to evaluate initialization of a pointer. This predicate holds if and only if the location pointed to by the pointer is considered initialized. It is defined as follows:

$$
\mathsf{init\_val}(v) \leftrightarrow \bigvee \begin{pmatrix} (v \in \mathsf{baseval} \wedge v \neq \mathsf{indet}\ \tau_b) \\ (v = \mathsf{struct}_t\ (\vec{v}) \wedge \mathsf{init\_val\_list}(\vec{v})) \\ (v = \mathsf{union}_t\ (i, v) \wedge \mathsf{init\_val}(v)) \\ (v = \overline{\mathsf{union}_t}\ (\vec{v}) \wedge \mathsf{init\_bit\_list}(\vec{v})) \end{pmatrix}
$$

$$
\mathsf{init\_val\_list}(\vec{v}) \leftrightarrow \forall v_n \in \vec{v}, \mathsf{init\_val}(v_n)
$$

$$
\mathsf{init\_bit\_list}(\vec{b}) \leftrightarrow \forall i, v_i \neq \lightning
$$

This definition has several cases, depending on which value is at the address pointed to by the pointer. If the value is a base value, we just need to verify that the value is not indeterminate. All the other base values are considered initialized. If the value is a structure, we verify that each value of each field is initialized. Unions are splitted into two cases since they have two representations. When the union has been initialized properly by one of its field, we just need to verify that the the value at this field is initialized. When the union has been initialized byte by byte, we don't know which field and which bits are useful, so we verify that every bit of this union has been initialized by flattening the value. A bit is considered initialized if its value is not indeterminate.

Finally, validity is based on danglingness of a pointer. After having evaluated the term, we verify that the resulting address is not dangling.

5.6 Statements

The statement assert($\wp$) links miniACSL with $\mathsf{CH_2O^-}$. Before we give the rule of this statement, we need to explain the semantics of $\mathsf{CH_2O}$'s statements. Program states are defined as tuples $S(P, \phi, m)$ where m is the (heap) memory and (P, ϕ) is a zipper-like data structure [21] that describes the part of the program that is being executed. P is a stack of all statement contexts that have been traversed since the start, while ϕ represents the current focus of the evaluation, as explained below. This complex structure is used to properly handle stack variables allocation and deallocation, as we enter and exit code blocks, especially in presence of non-structured control flow (i.e., `goto`).

Execution of statements is modeled by traversal through the zipper in several directions, corresponding to the different forms of control. This includes in particular:

- down ($\searrow$) indicates that we are in the process of evaluating the current statement;
- up ($\nearrow$) means that the current statement has been executed and we are ready to start the execution of the next statement in the sequence, popping the stack as needed to find a sequence;
- goto ($\rightsquigarrow l$) is used for finding the statement labelled by l.

The focus ϕ of the program state can be either a tuple (d, s) containing a direction and a statement, or an undefined focus defined as follows:

$$
\phi_u ::= \lightning e \mid \lightning S_e\langle v \rangle \mid \lightning \wp
$$

Undefined focus are useful when the execution encounter an undefined behavior of C (the two first cases, directly taken from $\mathsf{CH_2O}$) or when an assert has a predicate that we cannot prove to be true (the last case). These states are terminating the execution as there is no rule to continue from states being on this focus.

Reduction rules are identical in $\mathsf{CH_2O}$ and $\mathsf{CH_2O^-}$, except for assert($\wp$), which only exists in $\mathsf{CH_2O^-}$. Intuitively, if we are in a state where $\wp$ evaluates to True, assert($\wp$) acts as a skip, i.e. the execution proceeds, with the memory being left unchanged. Conversely, if $\wp$ evaluates to False, the execution stops. More formally, our rules for assert($\wp$) are thus the following.

$$
S(P, (\searrow, \mathsf{assert}(\wp)), m) \rightarrow S(P, (\nearrow, \mathsf{assert}(\wp)), m) \text{ if } \vDash [\![\wp]\!]
$$

$$
S(P, (\searrow, \mathsf{assert}(\wp)), m) \rightarrow S(P, \overline{\mathsf{undef}}(\lightning\wp), m) \text{ if } \nvDash [\![\wp]\!]
$$

It should be noted that, in the first case, we keep the assert in the focus, (as opposed to replacing it with a skip after successful evaluation), so that the standard stack unwinding rules will pick it up for a new evaluation on a new program state, should it become again under focus following a loop or a goto statement.

Let us illustrate this rule with the execution of the program from Fig. 3. We concentrate on the execution of the assert from line 13. We omit all lines that are not necessary for our example, so we only keep p1, p2 and p3 initializations and the assert itself. Hence, once the assert statement is in the stack as shown in the state $P1$ of Fig. 11, we need to evaluate the predicate that we want to assert to continue the execution.

$$
\begin{array}{lcl}
P1 & = & (x_2 := \mathsf{struct}_{p2D}\ (\mathsf{int}_{int}\ 5, \mathsf{int}_{int}\ 1); \Box) \\
 & & (local_{o_{p3}:\mathsf{struct}\ p2D}\ \Box) \\
 & & (x_1 := \mathsf{struct}_{p2D}\ (\mathsf{int}_{int}\ 2, \mathsf{int}_{int}\ 5); \Box) \\
 & & (local_{o_{p2}:\mathsf{struct}\ p2D}\ \Box) \\
 & & (x_0 := \mathsf{struct}_{p2D}\ (\mathsf{int}_{int}\ 2, \mathsf{int}_{int}\ 1); \Box) \\
 & & (local_{o_{p1}:\mathsf{struct}\ p2D}\ \Box) \\
\phi 1 & = & (\searrow, \mathsf{assert}(\wp)) \\
S_1 & = & S(P1, \phi 1, m) \\
\wp & = & x_0! = x_1 \wedge\ x_1! = x_2 \wedge x_0! = x_2 \\
[\![\wp]\!] & = & \mathsf{comp\_val}(!=, x_0, x_1) \overset{c}{\wedge} \\
 & & \mathsf{comp\_val}(!=, x_1, x_2) \overset{c}{\wedge} \\
 & & \mathsf{comp\_val}(!=, x_0, x_2)
\end{array}
$$

$$
\begin{array}{lcl}
P2 & = & (x_2 := \mathsf{struct}_{p2D}\ (\mathsf{int}_{int}\ 5, \mathsf{int}_{int}\ 1); \Box) \\
 & & (local_{o_{p3}:\mathsf{struct}\ p2D}\ \Box) \\
 & & (x_1 := \mathsf{struct}_{p2D}\ (\mathsf{int}_{int}\ 2, \mathsf{int}_{int}\ 5); \Box) \\
 & & (local_{o_{p2}:\mathsf{struct}\ p2D}\ \Box) \\
 & & (x_0 := \mathsf{struct}_{p2D}\ (\mathsf{int}_{int}\ 2, \mathsf{int}_{int}\ 1); \Box) \\
 & & (local_{o_{p1}:\mathsf{struct}\ p2D}\ \Box) \\
\phi 2 & = & (\nearrow, \mathsf{assert}(\wp)) \\
S_2 & = & S(P2, \phi 2, m)
\end{array}
$$

Figure 11: Example of assert Evaluation.

As said in Section 3.1, the statement $\mathsf{local}_\tau s$ is used to open a block scope with one local variable of type τ. So for the three variables p1 p2 and p3, a $\mathsf{local}_\tau s$ statement is used to declare each block for each variable. Each block scope has a variable x_i. The block of p1 uses the variable x_0, the one of p2 uses the variable x_1 and the last block, which is the one of p3, uses the variable x_2. At this point, once the predicate has been converted into a Coq Prop, we should give a proof of this Prop. If we succeed, it means that the predicate is true, so that we can continue the execution. As the predicate is provable here, we can continue the execution to state $S2$ by the rule of assert we introduced just before: as mentionned earlier, our assert statement behaves like a skip statement if the predicate in the assert is true, and is blocking execution otherwise.

6 CONCLUSION AND FUTURE WORKS

This paper presents a Coq formalization of a subset of the ACSL specification language for C code. It has focused on structures, unions and underspecified terms, which present specific challenges. Indeed, contrary to C, structures and unions are first-class values in ACSL. In particular, they may be compared. Underspecified terms and predicates are also ACSL first-class values. They include C expressions whose C semantics is an undefined behavior, such as `1/0 == (2-1)/(1-1)`. Providing a convenient semantics for these constructs is not easy, while critical for the language consistency.

Our development is based on the Krebbers' semantics of C [24]. It allows us to include the whole C language for free and to focus only on the specification language. We also benefit for free from his Coq development. In the future, it will help us to extend our work to a larger part of the ACSL specification language. ACSL includes several features that would be interesting to study, notably properties depending on specific program points, frame conditions and data dependencies, properties over floating-point values and real numbers, user-defined (recursive) logic functions, predicates and types, inductive and axiomatic definitions, logical arrays, and sets of values. On the longer term, this formalization work should also help testing existing code analyzers that evaluates ACSL predicates, typically the analyzers included in Frama-C, such as its abstract interpretation-based value analysis Eva [5], its deductive verification tool WP [1], and its runtime assertion checker E-ACSL [37].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments. This project was partly funded by project ANR-22-PECY-0005 "Secureval" managed by the French National Research Agency for France 2030.

REFERENCES

[1] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams. 2021. The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. *Commun. ACM* 64, 8 (2021).

[2] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. 2023. ACSL: ANSI/ISO C Specification Language. https://github.com/acsl-language/acsl/releases/download/1.19/acsl.pdf

[3] T. Benjamin and J. Signoles. 2023. Formalizing an Efficient Runtime Assertion Checker for an Arithmetic Language with Functions and Predicates. In *Symp. On Applied Computing (SAC)*.

[4] Lionel Blatter, Nikolai Kosmatov, Virgile Prevosto, and Pascale Le Gall. 2022. An Efficient VCGen-Based Modular Verification of Relational Properties. In *Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles (ISoLA)*. https://doi.org/10.1007/978-3-031-19849-6_28

[5] S. Blazy, D. Bühler, and B. Yakobowski. 2017. Structuring Abstract Interpreters through State and Value Abstractions. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. https://doi.org/10.1007/978-3-319-52234-0_7

[6] S. Blazy and X. Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* (2009).

[7] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. 2014. A trusted mechanised JavaScript specification. In *Symposium on Principles of Programming Languages (POPL)*.

[8] Denis Bogdănaș and Grigore Roșu. 2015. K-Java: A Complete Semantics of Java. In *Symp. on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2676726.2676982

[9] Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2024. Mechanizing Refinement Types. *Proc. ACM Program. Lang.* 8, POPL, Article 70 (jan 2024), 30 pages. https://doi.org/10.1145/3632912

[10] R. J. Boulton, A. D. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *Int. Conf. on Theorem Provers in Circuit Design: Theory, Practice and Experience*.

[11] P. Chalin. 2007. A Sound Assertion Semantics for the Dependable Systems Evolution Verifying Compiler. In *Int. Conf. on Software Engineering (ICSE '07)*. 23–33. https://doi.org/10.1109/ICSE.2007.9

[12] D. Chisnall, J. Matthiesen, K. Memarian, P. Sewell, and R. N. M. Watson. 2016. C memory object and value semantics : the space of de facto and ISO standard. (2016).

[13] L. A. Clarke and D. S. Rosenblum. 2006. A Historical Perspective on Runtime Assertion Checking in Software Development. *SIGSOFT Soft. Eng. Notes* (2006).

[14] C. Ellison and G. Roşu. 2012. An executable formal semantics of C with applications. In *Int. Symp. on Principles of Programming Languages (POPL)*.

[15] David Greenaway. 2014. *Automated proof-producing abstraction of C code.* Ph. D. Dissertation. https://api.semanticscholar.org/CorpusID:29734373

[16] R. Hähnle and M. Huisman. 2019. *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools.* https://doi.org/10.1007/978-3-319-91908-9_18
[17] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. 2012. Behavioral Interface Specification Languages. *Computing Surveys* 44, 3 (2012). https://doi.org/10.1145/2187671.2187678
[18] C. Hathhorn, C. Ellison, and G. Roşu. 2015. Defining the Undefinedness of C. In *Conf. on Programming Language Design and Implementation (PLDI)* (Portland, OR, USA) *(PLDI '15)*. 10 pages. https://doi.org/10.1145/2737924.2737979
[19] P. Herms. 2013. *Certification of a Tool Chain for Deductive Program Verification.* Ph. D. Dissertation. Université Paris-Sud.
[20] T. Hoare. 2003. The Verifying Compiler: A Grand Challenge for Computing Research. *J. ACM* 50, 1 (jan 2003), 63–69. https://doi.org/10.1145/602382.602403
[21] Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* (sep 1997). https://doi.org/10.1017/S0956796897002864
[22] M. Huisman and A. Wijs. 2023. *Runtime Annotation Checking.* https://doi.org/10.1007/978-3-031-30167-4_9
[23] ISO/IEC JTC 1/SC 22. 2011. *ISO/IEC 9899:2011 Programming languages — C.* Standard 9899:2011. ISO. https://www.iso.org/standard/57853.html
[24] R. Krebbers. 2015. *The C Standard Formalized in Coq.* Ph. D. Dissertation. Radboud University Nijmegen.
[25] D. Lazar, A. Arusoaie, T.-F. Serbanuta, C. Ellison, R. Mereuta, D. Lucanu, and G. Roş. 2012. Executing Formal Semantics with the K Tool. In *Int. Symp. on Formal Methods (FM).*
[26] G. T. Leavens, A. L. Baker, and C. Ruby. 1999. *JML: A Notation for Detailed Design.*
[27] J.-C. Léchenet. 2018. *Certified algorithms for program slicing.* Ph. D. Dissertation. Université Paris-Sud.
[28] H. Lehner. 2011. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking.* Ph. D. Dissertation. ETH Zürich.
[29] L. Li, Y. Liu, D. Postol, L. Lampropoulos, D. Van Horn, and M. Hicks. 2022. A Formal Model of Checked C. In *Computer Security Foundations Symp. (CSF).* https://doi.org/10.1109/CSF54842.2022.9919657
[30] D. Ly, F. Loulergue, N. Kosmatov, and J. Signoles. 2023. Sound Runtime Assertion Checking for Memory Properties via Program Transformation. *Formal Aspects of Computing* (2023).
[31] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. 2019. Exploring C Semantics and Pointer Provenance. https://doi.org/10.1145/3290380
[32] R. Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4
[33] K. Nienhuis, K. Memarian, and P. Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. In *Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/3022671.2983997
[34] M. Norrish. 1998. *Formalising C in HOL.* Ph. D. Dissertation. University of Cambridge.
[35] M. Norrish. 1999. Deterministic Expressions in C. In *Programming Languages and Systems.*
[36] D. Park, A. Ştefănescu, and G. Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Conf. on Programming Language Design and Implementation (PLDI).* https://doi.org/10.1145/2737924.2737991
[37] J. Signoles, N. Kosmatov, and K. Vorobyov. 2017. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. In *Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES).* https://doi.org/10.29007/fpdh
[38] Matthieu Sozeau, Abhishek Anand, Simon Pierre Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* 64 (2020), 947 – 999. https://api.semanticscholar.org/CorpusID:198315470
[39] Harvey Tuch. 2008. *Formal Memory Models for Verifying C Systems Code.* Ph. D. Dissertation. UNSW, Sydney, Australia.
[40] S. Zingg, S. Krstic, M. Raszyk, J. Schneider, and D. Traytel. 2022. Verified First-Order Monitoring with Recursive Rules. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).*

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009