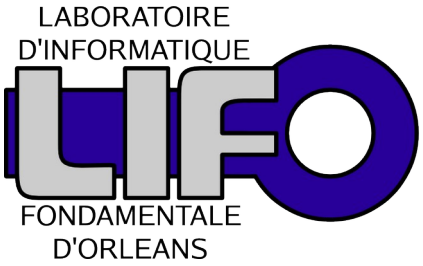

CoMeMoV: Collaborative Memory Models for formal Verification

Grant: ANR-22-CE25-0018

D2.a: Handbook of memory properties and memory models

Planned Date of Delivery	January 1rst, 2025
Actual Date of Delivery	December 12th, 2025
Deliverable Security Class	N/A
Author	Myriam Clouet
Proofreaders	Allan Blanchard, Frédéric Loulergue, Loïc Correnson, Nikolai Kosmatov

Contents

Executive summary

The objective of this report is to provide clear definitions of core concepts and properties for memory models in the selected semantics.

Introduction

This report is a *Handbook of memory properties and memory models*. The selected sources to establish the definitions are:

1. C standard [ISO11];
2. The memory model of the verified C compiler, CompCert [KLW14; LB08; Kre14; Ler+12; Ler+14];
3. The memory model proposed by Krebbers, CH_2O [Kre13; Kre16; Kre15];
4. The ACSL manual [Bau+23] and the chapter of ACSL in the Frama-C Book [BMP24];
5. The WP manual [Bau+24], the the chapter of ACSL in the Frama-C Book [Bla+24] and the WP tutorial [Bla20];
6. The memory model of the region analysis plugin of Frama-C [Dam+25];

As a first step this document focuses on two main operations in memory models: `load` and `store`. For that it also presents concepts and properties required for these operations. Similarities and differences between how the previously mentioned memory models are highlighted.

1 Values & Memory

1.1 Object

C standard Only the C standard defines objects. An object is a "region of data storage in the execution environment, the contents of which can represent values" [ISO11, §3.15]. The value of the object is the last stored value. An object has an address (constant during the execution), a type and a lifetime. Accessing an object outside its lifetime is an undefined behavior.

1.2 Value

C standard In our sources, only the one for the C standard provides a definition of a value: "precise meaning of the contents of an object when interpreted as having a specific type" [ISO11, §3.19]. Depending on the memory model, this value can be a logical/abstract representation (like an integer for ACSL), or a bit/byte representation (as for the concrete model of Krebbers).

1.3 Trap representation

C standard The C standard defines a trap representation as "an object representation that need not represent a value of the object type" [ISO11, §3.19.4].

1.4 Lvalues

C standard In the C standard, an lvalue is an expression (without a void type). This expression designates an object (see Section 1.1) (otherwise the behavior is undefined). The type of this object is the lvalue type. If the lvalue is the left operand of the assignment operator, it has to be a modifiable lvalue (the right operand could be called rvalue). The C standard details the computation of an lvalue type depending on its use with various operators. Depending on these criteria, an expression that has type "array of type" [ISO11, §6.3.2.1] could not be considered as an lvalue (and is converted to the type "pointer to type" [ISO11, §6.3.2.1]). There are type restrictions when accessing an object with an lvalue like type compatibility.

ACSL ACSL states to rely on the C standard definition of an lvalue. In its glossary it defines that an lvalue ("left-value" [Bau+23]) is a memory location represented in the code as a variable or an expression. It also considers modifiable and non-modifiable lvalues. ACSL also prints out that an lvalue is an expression "that can be used on the left of an assignment operator" [BMP24]. It presents examples like variable names, or structure, array or pointer accesses.

WP WP uses this term (or left-value and l-value) but does not define it. We can suppose it relies on the ACSL definition.

Region analysis The Region analysis, represents lvalues as memory locations, and cites some examples like variables, structure and union accesses but also the result of dereferencing operations and pointer arithmetic.

1.5 Aliasing

1.5.1 Definition

C standard The C standard does not define aliasing (or strict aliasing) but it lists "circumstances in which an object may or may not be aliased" [ISO11, §6.5]. This list corresponds to the allowed types of lvalues to access a stored value. Indeed, the C standard states "An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

- a type compatible with the effective type of the object,
- a qualified version of a type compatible with the effective type of the object,
- a type that is the signed or unsigned type corresponding to the effective type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type." [ISO11, §6.5]

CompCert CompCert provides semantics to undefined behaviors resulting from not respecting the C standard list of aliasing circumstances.

Krebbers Krebbers defines aliasing as "Aliasing is when multiple pointers refer to the same object in memory." [Kre13]. It also states that not respecting the C standard list of aliasing circumstances conducts to undefined behaviors.

WP For WP, alias is when "multiple pointers can have access to the same memory location" [Bla20]. In his definition, Krebbers uses the term "object" instead of "memory location". However, according to the C standard, an object is defined as a "region of data storage in the execution environment" [ISO11, §3.15]. Therefore, we may consider that the definition from WP is similar to Krebbers' definition.

Region analysis The region analysis of WP considers aliasing when expressions with pointers may potentially reference several different memory areas. Likely for the definition from WP, we may consider this definition is similar to the one from Krebbers.

1.5.2 Tackling aliasing

C standard The C standard provides rules to define when a pointer can or cannot alias.

CompCert CompCert states that aliasing has to be considered to allow program transformations. For example, for some operation transformations *nonaliasing* hypothesis or properties are required.

Krebbers To determine when a pointer aliases, Krebbers relies on a "type-based alias analysis" [Kre15]. He also states, for two pointers `p` and `q`, "a compiler should be able to assume that `p` and `q` are not aliased because they point to objects with different types" [Kre16] (e.g. if `p` is a int pointer and `q` a short pointer). Moreover, for Krebbers, if these pointers alias, the program should be considered to have an undefined behavior.

WP Our WP sources mention that ACSL annotations can create aliases in addition to the aliases already existing in the code. And they also mention WP assumes that ACSL annotations provide required information on potential variable aliasing, in order to perform its analysis. Besides the ACSL annotations, the WP memory models tackle the aliasing differently. The *Hoare* model cannot represent aliasing, whereas the *Type* model splits memory in different regions depending on the variable type. For example, "pointers to integers and to floating point values, they are assumed to refer to different memory regions" [Bla+24].

Region analysis To determine when a pointer aliases, Region analysis proposes to rely on an analysis by region. This analysis corresponds to a precise mapping of memory access. It considers that additional information is sometimes required to perform alias analysis. For example, it may be necessary to know the bound of an array to prevent an index to access an

address over this bound. Two regions are either disjoint, or one is included in the other. Two disjoint regions mean that their address ranges do not overlap.

1.6 Behaviors

C standard According to the C standard there are four types of behaviors:

- **Unspecified behavior -** "use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance" [ISO11, §3.4.4].
- **Implementation defined behavior -** "unspecified behavior where each implementation documents how the choice is made" [ISO11, §3.4.1]
- **Locale-specific behavior -** "behavior that depends on local conventions of nationality, culture, and language that each implementation documents" [ISO11, §3.4.2]
- **Undefined behavior -** "behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements" [ISO11, §3.4.3]

ACSL ACSL also provides definitions for these behaviors, except for the *Locale-specific behavior*. For some of its definitions, ACSL makes references to the C standard. It defines the *Unspecified behavior* as "where the standard is not precise, yet the behavior of the code in question will do something reasonable (and will never crash, that is: stopping execution abruptly) among the different possibilities" [BMP24]. It defines the *Implementation defined behavior* as "the execution is non-ambiguously defined, not from a standard (say the C11 standard [21]), but from implementation choices. [BMP24]. ACSL summaries the *Undefined behavior* as "where what will happen is not predictable, and the program could even crash" [BMP24].

CompCert CompCert summaries the definition of *Unspecified behavior* as "two or more behaviors are allowed" and moreover it considers that in this kind of behavior the "choice may vary for each use" [KLW14]. For *Undefined behavior*, CompCert presents it as "the standard imposes no requirements at all, the program is even allowed to crash" [KLW14], but it does not consider the *Locale-specific behavior*.

Krebbers Krebbers reuses the *Unspecified behavior* and *Undefined behavior* definitions from CompCert. Likewise, he does not consider the *Locale-specific behavior* either.

We may suppose that many of our sources, other than the C standard, do not provide a definition for the *Locale-specific behavior* because this behavior is mainly used in some C standard libraries, such as the "Character handling `<ctype.h>`" library [ISO11, §7.4]. Thus we may assume this behavior is usually not considered because this behavior not critical for other aspects of the standard.

1.7 Type representations

1.7.1 Basic Types

The C standard defines basic types like `int`, identifiers, `float` or `char`. These basic types exist in memory models; however in some memory models these basic types can also be represented using a bit/byte representation (as in CompCert [KLW14] or Krebbers [Kre13]) or with a corresponding abstract/logical type (as in ACSL [Bau+23]). CompCert uses a conversion operation to encode and decode values between byte representation and the basic type. The memory model of Krebbers keeps the abstract representation (e.g., `int` value) and the bit representation for each variable. To be noted that in ACSL, boolean and integer are different unlike in C standard. This implies an implicit cast when boolean operators are used with integers in the program [Bau+23].

1.7.2 Aggregate type representations

The C standard provides a definition of arrays: "An array type describes a contiguously allocated nonempty set of objects with a particular member object type, called the element type." [ISO11]. Besides, the C standard considers that the element type and the number of elements in the array characterize the array type. In C standard, pointers and arrays are strongly linked. Indeed, "a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type" [ISO11] and "E1[E2] is identical to (*((E1)+(E2)))" [ISO11].

ACSL defines specific logic types to map C aggregate types, and their fields are mapped recursively [Bau+23]. In the logical language $\mathcal{L}$ of ACSL, a structure is represented as a *record* and an array as a total *map* [BMP24; Bau+24].

In our sources for CompCert [KLW14; LB08; Kre14], Krebbers [Kre13], and for Region analysis [Dam+25] there is no definition or explanation of

array representations. So we could suppose that they rely on the C standard definition.

For the C standard, structures and arrays are aggregate types unlike unions, because a union "can only contain one member at a time" [ISO11, §6.2.4, note (46)]. However structures and unions share many characteristics and are defined together in the C standard. For example, "each structure or union has a separate name space for its members" [ISO11, §6.2.3]. Besides ACSL considers that unions are aggregate types [Bau+23] (and they can have corresponding logic types). This is also the case for the Region analysis [Dam+25]. Therefore in this document we also consider that unions are aggregate types.

Many constraints and properties are defined on structures and on unions in the C standard. Like "type punning" [ISO11, §6.5.2.3, note (95)], that consists to read a different union field than the last written. This conducts some memory models to adapt their semantic to prevent undefined behavior. For example, CompCert memory model keeps the value of a variable but also its type (i.e. the type can be updated in the memory like the value) [LB08]. Another example, Krebbers defines two kinds of union, one specific when there was a byte-wise copying [Kre13].

For structures, WP uses logical offsets for their elementary types instead of the real offsets (based on the size and alignment of fields) [Bla+24].

1.7.3 Pointer representations

Simple pointers – The C standard defines pointers as: "A pointer type describes an object whose value provides a reference to an entity of the referenced type" [ISO11]. A pointer can reference an object (even another pointer) or a function. The standard defines many rules to the alignment in the memory (for example, depending on the type of the pointer or the operations on the pointer). It also defines rules to correctly cast pointers and to perform arithmetic on pointers. The arithmetic on pointers is usually used for access array elements.

In CompCert, pointers are represented as a pair (b, i) where b is the "block" (memory location) and i is "the offset in that block" [KLW14]. The value of a pointer correspond to "symbolic bytes" and a block is an array of "symbolic bytes" [KLW14].

In ACSL, pointers are memory locations. These memory locations are separated by blocks. The ACSL operation `\base_addr{L}(p)` "returns the base address of the allocated block containing, at the label `L`, the pointer `p`" [Bau+23]. And the ACSL operation `\offset{L}(p)` "returns the offset in bytes between `p` and its base address" [Bau+23]. Therefore we can translate this representation of pointer as a pair of (base, offset) like the CompCert representation. It is specified in our

ACSL sources [BMP24; Bau+23], that many properties and specificities of pointers in ACSL (i.e. in ACSL annotations). For example, "`\at(*p,L)` is usually different from `*\at(p,L)`" [BMP24]. The former means the value referred to by the pointer `p` is evaluated at the label `L`. The latter means the address of `p` is evaluated at the label `L` but the dereferencing (to get the pointed value) is evaluated where the `\at` is written.

In WP/region analysis [Dam+25], pointers are addresses. These addresses are regrouped in regions of the region map for the analysis.

End-of-array pointers/one past the last element pointers – As seen previously, in the C standard, the arithmetic on pointers is usually used for array access. A specific case is considered: the last element of an array +1. The result of this operation is named "one past the last element of the array object" [ISO11]. This will not overflow but it should not be used to read or write the value (it only can be used in arithmetic on pointers).

CompCert tackles "end-of-array pointers" [KLW14] since the version 1.13. These pointers correspond to the "one past the last element of the array object" of the C standard. As for the C standard, these pointers cannot be dereferenced (read) but they can be used in arithmetic on pointer as for example when looping through arrays. In [KLW14], a difference between compilers is highlighted: for some compilers, when two variables are declared, one after the other, their addresses are consecutive in the stack. Therefore incrementing the address of the first variable correspond to the address of the second. It is not the case in CompCert because the addresses of the variables are in different blocks. To our knowledge, this behavior is not defined in the C standard and even could be considered as an undefined behavior.

In his memory model, Krebbers considers the end-of-array pointers. He defines various types of pointer, the type *References* "can describe most pointers in C but cannot account for end-of-array pointers and pointers to individual bytes". For these specific pointer types, he defines another type, *Address*, that extends the *Reference* type. But he restricts the use of end-of-array pointers: "we assign undefined behavior to questionable uses of end-of-array pointers while assigning the correct defined behavior to pointer comparisons involving end-of-array pointers when looping through arrays" [Kre15]. For example, a "questionable" use may be the comparison of supposed objects adjacent in memory.

```
1 int x, y;
2 if (&x + 1 == &y) printf("x␣and␣y␣are␣
    adjacent\n")
```

"x and y are adjacent" is printed when compilers allocate `x` and `y` consecutively on the stack, which is often, but not necessarily the case [KLW14].

There is no mention in our ACSL sources [BMP24; Bau+23] neither in our WP and WP/region analysis sources [Bau+24; Bla+24; Bla20; Dam+25] for this type of pointers.

Function pointers – In C standard [ISO11], when a function pointer is dereferenced it corresponds to the *function designator*. Conversion between function pointers is allowed but the types have to be compatible.

ACSL allows to use function pointers but restricts this use only for equality checks [Bau+23]. A specific predicate is defined in ACSL to check if a function pointer is valid [Bau+23].

When function pointers are used, WP performs various verification on the pointed function depending on ACSL specifications and the context of the use.

There is no mention of function pointers in our sources for CompCert [KLW14; LB08; Kre14], Krebbers [Kre13], and in Region analysis [Dam+25]. We could suppose that they do not consider them or they are considered like other pointers.

1.7.4 Special types

The C standard defines a specific type, the **void type** that is "an incomplete object type that cannot be completed." [ISO11]. However, contrary of other types, the value of an expression that has this type "shall not be used in any way", even for conversion (except to void). However Krebbers explains its role: "it is used for functions without a return value, and for pointers to data of an unspecified type." [Kre13] The memory model of Krebbers also differs from the C standard: According to Krebbers, in the C standard, the type "`void` is used for functions without return type and `void*` is used for pointers to objects of unspecified type" [Kre16]. In his model, Krebbers retains the usage of the type `void` but he changes the type notation `void*` to a new type notation `any *`.

In CompCert [LB08], a specific type (for constants) is defined to represent an undefined value (for example for an uninitialized variable): the **undef type**.

1.8 Memory

1.8.1 Scope

A scope for the C standard [ISO11] is space inside which an identifier can only designates a same entity. In this scope the identifier is considered as "visible" and therefore can be used. There are four types of scope: function, file, block, and function prototype [ISO11]. The scope type of the identifier is determined by the placement of its declaration. For example, when "the identifier appears outside of any block or list of parameters, the identifier has file scope" [ISO11].

CompCert does not provide a definition of a scope but it relies on the "block scope" term for its variable analysis [Kre14]. Beside scope, CompCert uses *Program context*, a data structure storing "the location of the substatement that is being executed" [Kre14].

There is no scope definition in our sources for ACSL or for WP. However ACSL provides additional scopes to the C standard like the "function contract scope" [Bau+23]. Besides, some of its specifications rely on the C standard scope to infer information. For example, "when no `assigns` clause is given then any global variable in the scope of the considered function might be modified" [BMP24], In our sources for WP, some cases when scope has to be carefully considered are mentioned. For example, for dynamic allocation [Bau+24] or when using the `\at` annotation, scope has to be respected [Bla20]. WP and ACSL also rely on scope to define dangling addresses and pointers [Bau+23; Bla+24].

There is no mention of scope in our sources for Region analysis [Dam+25] and for Krebbers [Kre13].

1.8.2 Block

A block is a syntactic unit that allows to group a set of declarations and statements [ISO11]. CompCert considers a notion of a sub-block during its program transformations [LB08].

There is no block definition in our ACSL, WP and CompCert sources however, they use this term to define their memory representations. One has to be careful when using the term *block* to be sure if it represents a "set of declarations and statements" [ISO11] or a part of the memory [KLW14; BMP24; Bla+24]. For example, WP refers to *program* blocks (like "block of the loop") [Bla20] and to *memory* blocks [Bla+24]. These kinds of blocks correspond to the C standard definition (and not to a memory representation). WP also defines specific types of blocks (for example, "*axiomatic* blocks" [Bla+24]). As in ACSL, a *memory* block is "characterized by its base address and its length" [BMP24], ACSL provides functions to manipulate *memory* blocks and to get information, like its base address [BMP24].

There is no mention of blocks in our sources for Krebbers [Kre13] and for Region analysis [Dam+25].

1.8.3 Memory representations

The C standard The C standard does not provide a definition of its memory representation but it provides a definition of a memory location. A memory location is "either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width" [ISO11]. For reminder, scalar types are arithmetic types and pointer types [ISO11].

CompCert CompCert has an abstract and a concrete memory model. Both rely on blocks, a block being an array of symbolic bytes [KLW14] that can be "addressed using byte offsets $i \in \mathbb{Z}$" [LB08]. Memory states store values but also their types (as type-value pairs (τ, v)) [LB08]. The memory state of the abstract model is only a collection of separated blocks, identified by block references b [LB08]. The memory state of the concrete model is more complex, it is composed of different elements. It is a "quadruple (N, B, F, C), where

- N : `block` is the first block not yet allocated;
- B : `block` $\to \mathbb{Z} \times \mathbb{Z}$ associates bounds to each block reference;
- F : `block` $\to$ `boolean` says, for each block, whether it has been deallocated (`true`) or not (`false`);
- C : `block` $\times\mathbb{Z} \to$ `option` (`memtype` $\times$ `val`) associates a content to each block b and each byte offset i. A content is either ε, meaning "invalid", or $\lfloor \tau, v \rfloor$, meaning that a value v was stored at this location with type τ" [LB08].

In CompCert, a memory location is a "pair (b, i) of a block reference b and an offset i within this block." [LB08]

ACSL ACSL considers two separated memories: a C memory (for the C variables) and a logical memory (for logical variables). The values in the latter do not have addresses. Like in CompCert, the C memory model of ACSL is composed of memory blocks. A memory block is "characterized by its base address and its length" [BMP24]. A base address is either "the address of the declared object" or "the pointer returned by an allocating function" [BMP24].

Krebbers Unlike CompCert, the memory model of Krebbers does not use bytes but bits [Kre13]. Another difference, Krebbers does not represent the memory as a collection of blocks (array of bytes) but instead it represents the memory as trees [Kre13]. Its memory model has three *layers*: one for abstract values, one for memory values and one as an array of bits. The first two are trees but their leaves are different. The leaves of the tree for abstract values are mathematical integers and pointers whereas the leafs of the tree for memory values are arrays of bits [Kre13]. The model of Krebbers can tackle alignment thanks to the function `fieldsizes` that has to be set depending on the wanted alignment. The memory does not only store values but, like for CompCert, it also stores the type of the stored value. The memory is formalized as following [Kre15]:

$$m \in \texttt{mem} := \texttt{index} \rightarrow_{fin} (\ \texttt{mtree} \times \texttt{bool} + \texttt{type}\ )$$

$$\begin{aligned} w \in \texttt{mtree} ::= {} & \texttt{base}_{\tau_b}\vec{\mathbf{b}} \mid \texttt{array}_{\tau}\,\vec{w} \mid \texttt{struct}_t\,\overrightarrow{w\mathbf{b}} \\ & \mid \text{union}_t\,(i, w, \vec{\mathbf{b}}) \mid \overline{\text{union}_t}\,\vec{\mathbf{b}} \end{aligned}$$

In memory, each object is annotated with a boolean value and with its type. The boolean corresponds to information about the allocation. If the object has been allocated using `malloc`, the boolean is `true`. Krebbers distinguishes between two kinds of unions, $\text{union}_t\,(i, w, \vec{\mathbf{b}})$ and $\overline{\text{union}_t}\,\vec{\mathbf{b}}$. The former corresponds to a union with a known variant, i. These unions can only be accessed through a pointer to this variant. The latter corresponds to a union with a currently unspecified variant.

WP WP has different memory model definitions. A memory model is "a set of datatypes, operations and properties that are used to abstract the values living inside the heap during the program execution" [Bau+24], and it describes "how the memory behaves when some assignments are performed by the program" [Bla+24]. But a memory model also "defines a mapping from values inside the C memory heap to mathematical terms" [Bau+24]. WP provides several memory models in order to enable to use of the simplest memory model when it is possible.

- The *Hoare* model : the simplest model. It "maps each C variable to one pure logical variable." Some hypotheses are defined [Bau+24]:
  - the variables are separated one form another;
  - the variables are separated from the addresses.

  Pointers can be represented with this model but not read or written [Bla20].
- The *Reference* model: this model can detect if a program with pointers as function parameters can be modeled with the *Hoare* memory model. For that it requires some conditions for the pointers as parameters [Bla+24]:

- they are never modified (the pointers, not the pointed values);
- they are separated;
- they point to valid locations.

– The *Typed* model : the default memory model of WP. This model contains memory variables to represent the heap for basic types (char, int32, unsigned int, floats32, float64, ...) and pointers [Bla20]. These variables are arrays indexed by addresses. In WP, an address is a pair (base,offset). This offset is a logical integer (so it is not bytes). Another array is defined to represent the memory allocation ($base \rightarrow length$) and another to represent the initialization ($address \rightarrow bool$) [Bau+24]. The basic type values are logical values. The pointer values in this memory are indices of the arrays. Variables in different memory variables are considered as *separated*. This model also "assumes that all pointers may alias" [Bla+24]. Heterogeneous casts cannot be represented with this memory model but some casts are authorized.

– The *Caveat* model : this model is based on the `Caveat` analyzer. It is an extension of the *Typed* model but with a specific treatment for the formal and references parameters. Unlike the *Reference* model, pointers as parameters can be modified in the function. But this model assumes that there is no aliasing between the pointers received in formal parameters, the user has to verify this hypothesis [Bau+24].

– The *Bytes* model: a low-level memory model [Bla20]. The memory is represented by a wide array of bytes and pointer values are addresses [Bau+24]. This impacts read and write operations which become manipulations on range of bits. This model is the more precise of the WP models but it generates complex proof obligations which are difficult to discharge by automated provers and may require the use of an interactive proof assistant [Bla20].

Besides these memory models, Frama-C assures that ghost code is separated from C code [Bla20]. Therefore ghost code can only modify the ghost memory.

Region analysis The work on the region map analysis aims to define a new efficient memory model for WP but currently there is no precise modeling of the memory in [Dam+25].

2 Operations

The operations described in our sources may differ by their definitions but also by their types, some operations may be present in certain sources but

not in others. We choose to provide the description of the operations even if they are not present in all of our sources. We do not indicate when they are missing, but if we do not mention the sources this means that those sources do not describe the concerned operation.

2.1 Aggregate copy

For the C standard, to copy an aggregate type, it is necessary to use the `memcpy` (or `memmove`) function, a simple assignment does not work as a *real* copy on aggregate types. For example, with two structs `s1` and `s2`, the operation `*s1 = *s2` does not copy all fields from `s2` to `s1`. This operation only copies the first member of the structure. These functions could have side effects (not just copy the value). For example, "If copying takes place between objects that overlap, the behavior is undefined" [ISO11, §7.24.2]. Another example, depending on certain conditions, the effective type of the modified object could change.

In ACSL [BMP24], an implementation of the function `memcpy` is used to illustrate the necessity to check the pointer separation (with the predicate `separated`).

In [Kre13], Krebbers demonstrates the use of its memory model for formal proof by illustrating it on an abstract version of the `memcpy`.

2.2 Support for pointer operations

To avoid invalid access to the memory though pointers, ACSL provides various predicate to verify the status of a pointer and if these operations are possibles. For example: `\allocable` `\freeable`, or `\dangling`. There is a difference between ACSL and the C standard: `\freeable(\null)` does not hold even if in the C standard the free operation is possible on the null pointer (it is a valid argument to the function `free`) [Bau+23].

By default, for WP, pointers do not have a particular validity status. The user can provide this information, using the `\valid` annotation. However, in its *Ref* memory model, pointer validity is assumed when dereferencing pointer [Bla+24].

To allow byte-wise reading and writing, CompCert [KLW14] defines a specific constructor (`Vptrfrag`).

2.3 Operations with undefined behavior

Some operations are considered as an undefined behavior:

- Dereferencing a NULL or dangling pointer [KLW14];
- Use of dangling block-scope pointers [KW13];
- Adding 0 to a byte from a pointer object representation [KLW14];

– Arithmetical operations on pointers fragments [KLW14];

– Accessing to a union field with a type different than the previous one used to store the value [LB08];

– Accessing an array out of its bounds [BMP24; KLW14; Kre14].

– Read a variable non-initialized [Bla+24];

– Writing to a non-properly allocated memory location [BMP24];

– The order and contiguity of storage allocated by successive calls to the calloc, malloc, and realloc functions [ISO11];

– Loading from a freshly allocated block (with the concrete model of CompCert, choice made by CompCert) [LB08]

– Loading just after a store with an incompatible type (transformed from unspecified to undefined behavior in the concrete model of CompCert) [LB08]

– Loading just after a store with a different type in the same location (transformed from unspecified to undefined behavior in the concrete model of CompCert) [LB08]

– The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to `free` or `realloc`(with the concrete model of CompCert, choice made by CompCert) [LB08].

Some memory operations have an **unspecified behavior**:

– Loading from a freshly allocated block [LB08];

– Loading from a freshly deallocated block [LB08];

– Loading just after a store with an incompatible type [LB08];

– Loading just after a store with a different type in the same location [LB08]

– The pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to `free` or `realloc` [LB08].

The differences between undefined and unspecified behaviors are explained in Section 1.6.

2.4 Block/memory allocation

C standard [ISO11] For the allocation of the memory, the C standard provides different functions, `aligned_alloc`, `calloc`, `malloc`, and `realloc`. They are used in different situation. For example, the `malloc` function is used when allocating an object with a known size but an unknown value, and the `realloc` function is used when deallocating an old object to reuse its memory location for a new object. The C standard specifies many rules about allocation. For example:

- the "pointer returned if the allocation succeeds is suitably aligned";
- an "allocation shall yield a pointer to an object disjoint from any other object";
- the "lifetime of an allocated object extends from the allocation until the deallocation";
- the "pointer returned points to the start (lowest byte address) of the allocated space";
- "If the space cannot be allocated, a null pointer is returned";
- "If ptr is a null pointer, the realloc function behaves like the malloc function for the specified size'.'

It also specifies when an allocation has an implementation defined behavior or an undefined behavior. Like when the size of the space requested is zero (implementation defined) or when the `realloc` function is used on a deallocated pointer (undefined).

CompCert CompCert has two memory models: an abstract model and a concrete model. When we do not mention the difference in our explanations, it means that the difference is not mentioned in our sources, therefore we assume that the explanations are valid in both models. CompCert reasons with block allocation and not physical addresses. Therefore it is not possible to perform arithmetic on physical addresses. For example, when two variables are consecutively declared, they are not adjacently allocated in the memory, unlike in some compiler, therefore an incremental address of the former is not equal to the latter [KLW14].

The allocation function is defined in CompCert [LB08] as:

$$\texttt{alloc: mem} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \texttt{option (block} \times \texttt{mem)}$$

This function takes the memory state, the low bound and high bound for the fresh block and then returns the reference of the new block and the updated memory state. The low bound is inclusive whereas the high bound is

not [LB08]. The allocation function can fail when there is not enough available memory. However in the abstract model of CompCert, the memory is infinite. Unlike in the C standard, for example with the `realloc` function, the allocation function in CompCert always allocates new blocks, i.e. blocks are not reused (event if they were deallocated before) [LB08]. In the concrete model of CompCert, the allocation is deterministic with respect to the domain of the current memory state (the block contents may differ).

In the CompCert version 2, this operation is redefined as:

$$\texttt{alloc: mem} \to \mathbb{Z} \to \mathbb{Z} \to \texttt{mem} \times \texttt{block}$$

The CompCert version 2 states "`alloc`, `load`, and `store` are as in version 1 of the model" [Ler+12], but as we can see, there is no further mention of the `option`. We could suppose this operation cannot fail anymore. Moreover, in version 1, this operation relies on the `low_bound` and `high_bound` of a block but in version 2, these are removed and replaced by "byte-level permissions" [Ler+12]. Sorted from the most to the least permissive, the permissions are: `Freeable`, `Writable`, `Readable`, `Nonempty`. Every byte location is associated with permissions: the current one and the maximal one. Therefore, the mechanism of block allocation has changed.

A new operation on memory is added in CompCert version 2 to tackle these permissions:

$$\texttt{drop\_perm : mem} \to \texttt{block} \to (Z) \to (Z)\ \texttt{permission} \to \texttt{option mem}$$

It allows lowering the permissions over locations in the block.

The `alloc` from CompCert does not exactly correspond to the `malloc` function in the C standard, it models "the creation of fresh blocks at function entry" [Ler+14] and "could also be used to model acquisition of address space via system calls" [Ler+14].

Krebbers In his memory model, he defines an allocation operation :

$$\texttt{alloc}_\Gamma\ \texttt{:}\ \ \texttt{index} \to \texttt{val} \to \texttt{bool} \to \texttt{mem} \to \texttt{mem}$$
$$\texttt{alloc}_\Gamma o\ \nu\ \mu\ m := m[o{:=}(\texttt{ofval}_\Gamma\ (\Diamond(0,\ 1)^{\texttt{bitsizeof}_\Gamma(\texttt{typeof}\nu)})\ \nu,\ \mu)]$$

This operation does not rely on blocks or indices in memory. An allocation, $\texttt{alloc}_\Gamma\ o\ \nu\ \mu\ m$, associates a unique identifier ($o \notin \texttt{dom}\ m$), with the value ν in the memory m. The boolean μ enables to indicate if this allocation is performed using `malloc` (thus $\mu = \top$), this allows to represent dynamic allocation. This operation relies on the function `dom` that associates memory indices (`index`) with the memory (`mem`): $\texttt{dom :}\ \ \texttt{mem} \to \mathcal{P}_{fin}(\texttt{index})$. These indices are from a countable set. The nature of this set is not defined because "since we do not rely on any properties apart from countability, we keep the representation opaque" [Kre15].

ACSL [Bau+23] In ACSL, it is possible to specify a behavior about allocation, with the clause `allocates`. If it is not used, it means that there is no newly allocated memory block.

ACSL also provides some predicates to support reasoning about allocation. These predicates are presented in Section 2.2.

2.5 Memory initialization

C standard The C standard defines many rules how to initialize variables. For example : "The type of the entity to be initialized shall be an array of unknown size or a complete object type that is not a variable length array type" [ISO11, §6.7.9]. It also defines specific rules to define the default initializations when no value are given. For example : "if it has pointer type, it is initialized to a null pointer" [ISO11, p. 6.7.9] and "if it has arithmetic type, it is initialized to (positive or unsigned) zero" [ISO11, p. 6.7.9].

CompCert CompCert represents uninitialized memory by symbolic bytes. It also defines a specific constant value, `undef`, to represent the value of uninitialized variables in the memory and a constant, `empty`, to represent the initial memory state. [LB08] In its concrete model, the value returned for a variable in the empty state is ε, meaning "invalid".

ACSL ACSL provides a predicate,`\initialized`, to check if a set of lvalues (*e.g.* pointers) are initialized at a specific program label [Bau+23].

WP The memory models in WP keep a specific variable to represent the variable initialization which indicate for every program variable if it is initialized or not [Bla+24].

2.6 Memory access

Two main memory operations are `store` a value and `load` a value. In some memory definitions they could be named "write", "modify", "assign" or "read" and even simply "access".

C standard [ISO11] In C standard, these operations are not formally described but many properties and specific cases are considered. These operations are strongly related to the type of the value accessed. The `store` operation corresponds to the assign expression (`=`), it stores a value resulting from the expression (as rvalue) in an lvalue. This could result in a *trap representation* (cf Section 1.3).

CompCert [KLW14; LB08; Kre14] The `load` and `store` operations are defined twice in CompCert: for the abstract model and for the concrete model.

For the abstract model, the `load` operation in CompCert gets a value from a memory block (that could be shifted according to an offset). This operation also considers the type of the read data. For the concrete model, the `load` operation performs the same action and besides it checks several conditions: if the read address and the read value are valid, if the types are compatibles and if the previous stored data at this address was not partially overwritten. In both models, this operation is defined as:

$$\texttt{load: memtype} \times \texttt{mem} \times \texttt{block} \times \mathbb{Z} \rightarrow \texttt{option val}.$$

For the abstract model, the `store` operation writes the value in a memory block (that could also be shifted according to an offset) and returns the updated memory if the operation is successful. In both models, this operation is defined as:

$$\texttt{store: memtype} \times \texttt{mem} \times \texttt{block} \times \mathbb{Z} \times \texttt{val} \rightarrow \texttt{option mem}.$$

For the concrete model, the `store` operation performs the same action and more. It checks the validity of the address. It also erases partial data that could remain in the memory block after writing the data depending of the size of the data type.

When a value is stored, then this value can be loaded only with the same type or with a compatible type (otherwise the load will fail). If the types were different but compatible the read value is converted to the wanted loaded type.

CompCert considers that a load operation has no side effect on the memory.

In version 2, CompCert redefines these operations and introduces two additional ones for byte contents. The `load` operation is redefined as follows:

$$\texttt{load: mem} \rightarrow \texttt{memory\_chunk} \rightarrow \texttt{block} \rightarrow \mathbb{Z} \rightarrow \texttt{option val}.$$

And the `store` operation is redefined as:

$$\texttt{store: mem} \rightarrow \texttt{memory\_chunk} \rightarrow \texttt{block} \rightarrow \mathbb{Z} \rightarrow \texttt{val} \rightarrow \texttt{option mem}.$$

Even if it states that "`alloc`, `load`, and `store` are as in version 1 of the model" [Ler+12], they are slightly different. The function `memory_chunk` does not exist in version 1. `memory_chunk` is not formally defined in this source [Ler+12], but relying on the definition of the `encode_val`, we may consider it to be the type of the values (i.e. `Mint32`). According to this observation, we may conclude that it corresponds to the `memtype` from version 1. Moreover, these operations checked the bounds in version 1, but in version 2, they rely on the permissions. They check if the accessed locations are associated with `Writable` and `Readable` for their current permissions.

The additional operations are :

$$\texttt{loadbytes}: \ \texttt{mem} \to \texttt{block} \to \mathbb{Z} \to \mathbb{Z} \to \texttt{option (list memval)}.$$

And

$$\texttt{storebytes}: \ \texttt{mem} \to \texttt{block} \to \mathbb{Z} \to \texttt{list memval} \to \texttt{option mem}.$$

These operations correspond to the `load` and `store` operations for reading and writing a list of byte contents. We may suppose that these operations also check the permissions of the accessed locations.

In version 2, the `load` and `store` operations work like `loadbytes` and `storebytes` but use `decode` and `encode` functions.

Krebbers [Kre13]: In [Kre13], Krebbers defines 4 memory operations: `alloc`, `free`, `_!!_` (corresponding to a `load`), and `_[_:=_]` (corresponding to a `store`). The `load` operation, gets a value from a specific address in the memory, that could fail if the stored value does not exist in the memory or if the types are not respected. The `store` operation writes a value at a specific address in the memory.

Krebbers defines these operations twice: on abstract values (e.g. integers) and on byte representations of the values.

For the `load` operation on abstract values, he considers various cases depending on the type of the reference: if it is a simple reference, an array, a structure, or a union. For the union type, he also considers if the load is performed with the same *variant* (field) that was used for the previous storage.

When this `load` operation is performed on an array, a structure, or a union, the shift is computed to get the value at the requested address. For example, to get the ith value of an array, the `load` operation will compute the corresponding address from the array reference and the offset i. This operation is defined as:

$$\texttt{\_!!\_}: \ \texttt{mval} \to \texttt{ref} \to \texttt{mval}^{\texttt{opt}}.$$

The `load` operation on byte representations considers two cases: when the whole abstract value is got or when it is a specific byte of the abstract value. This operation is defined as:

$$\texttt{\_!!\_}: \texttt{mem} \to \texttt{addr} \to \texttt{val}^{\texttt{opt}}.$$

Krebbers also considers a specific memory operation (called `force`) to tackle the side-effects of a memory access in order to respect the strict aliasing. This operation changes the effective types after a succeeded `load`.

The `store` operation can be performed only if the address is accessible, which is the case if the `load` operation (`_ !! _`) succeeds. This operation is not defined in [Kre13] but we suppose it could be defined as:

$$\texttt{\_[\_:=\_]}: \text{mem} \to \text{ref} \to \text{val} \to \text{mem}.$$

On byte representations This operation depends on the `load` operation to get bytes from the stored value. This operation is defined as:

$$\texttt{\_[\_:=\_]}\colon \text{mem} \to \text{addr} \to \text{val} \to \text{mem}.$$

In his thesis, Krebbers defines another `load` operation :

$$\begin{gathered} \mathit{lookup}_\Gamma : \texttt{addr} \to \texttt{mem} \to \texttt{option val} \text{ (with } m\langle a\rangle_\Gamma := \mathit{lookup}_\Gamma a m) \\ m\langle a\rangle_\Gamma := \texttt{toval}_\Gamma w \ \texttt{if}\ m[a]_\Gamma = w \ \texttt{and}\ \forall i.\texttt{Readable} \subseteq \texttt{kind}\ (\bar{w})_i \end{gathered}$$

This operation gets the value stored at the address a in the memory m. It could fail for various reasons: "if the permissions are insufficient, effective types are violated, or a is an end-of-array address" [Kre15]. This operation may have side effects on the effective type of the stored value.

This operation relies on the function `force` to represent this side effect:

$$\begin{gathered} \texttt{force}_\Gamma : \ \texttt{addr} \to \texttt{mem} \to \texttt{mem} \\ \texttt{force}_\Gamma\ a\ m\ \texttt{:=}\ m[(\texttt{index} a) := (w[\texttt{ref}_\Gamma a/\lambda w'.w']_\Gamma, \mu)] \end{gathered}$$

$\texttt{if}\ m(\texttt{index}\ a) = (w, \mu)$ This function "does not change the memory contents, but merely changes the variants of the involved unions" [Kre15].

In his thesis, Krebbers also provides another definition for the `store` operation:

$$\begin{gathered} \texttt{insert}_\Gamma : \ \texttt{addr} \to \texttt{mem} \to \texttt{val} \to \texttt{mem} \text{ (with } m\langle a := \nu\rangle_\Gamma := \texttt{insert}_\Gamma\ a \\ \nu\ \mu) \\ m\langle a := \nu\rangle_\Gamma := m[a/\lambda w.\ \texttt{ofval}_\Gamma(\bar{w}_1)\nu]_\Gamma \end{gathered}$$

This store operation succeeds only under certain conditions: "in case permissions are sufficient, effective types are not violated, and a is not an end-of-array address" [Kre15]. The function `writable` enables the detection of these conditions:

$$\begin{gathered} \texttt{writable}_\Gamma : \ \texttt{addr} \to \texttt{mem} \to \texttt{Prop} \\ \texttt{writable}_\Gamma\ a\ m\ \texttt{:=}\ \exists w.m[a]_\Gamma\ \texttt{=}\ w\ \texttt{and}\ \forall i.\texttt{Writable} \subseteq \texttt{kind}\ (\bar{w})_i \end{gathered}$$

Krebbers defines a permission mechanism to meet the *sequence point restriction* requirement based on `lock` and `unlock` functions :

$$\begin{gathered} \texttt{lock}_\Gamma : \ \texttt{addr} \to \texttt{mem} \to \texttt{mem} \\ \texttt{lock}_\Gamma\ a\ m := m[a/\lambda w.\ \text{apply } \texttt{lock} \text{ to all permissions of } w]_\Gamma \end{gathered}$$

$$\begin{gathered} \texttt{unlock} : \ \texttt{lockset} \to \texttt{mem} \to \texttt{mem} \\ \texttt{unlock}\ \Omega\ m := \{(o, (\hat{f} w \vec{y},)) \mid m\ o = (w, \mu)\} \cup \{(o, \tau) \mid m\ o = \tau\} \\ \text{where } f(\gamma, b)\ \texttt{true} := (\texttt{unlock}\ \gamma, b) \\ f(\gamma, b)\ \texttt{false} := (\gamma, b), \\ \text{and } \vec{y} := ((o, 0) \in \Omega) \ldots ((o, |\texttt{bitsizeof}_\Gamma(\texttt{typeof}\ w)| - 1) \in \Omega) \end{gathered}$$

When a `store` is performed, the stored object is *locked* to prohibit access until the next sequence point, at which the object is *unlocked*. Another permission kind, the `Locked`, is defined to meet the *sequence point restriction* requirement. The *sequence point restriction* "assigns undefined behavior to programs in which an object in memory is modified more than once in between two sequence points" [Kre15].

This permission mechanism is also used for other memory operations (e.g., allocation). There are various kinds of permissions. From the most to the least permissive: `Writable`, `Readable`, `Existing`, and `Empty`. The `Existing` kind is used when a pointer can only be used for pointer arithmetic operations.

ACSL [BMP24; Bau+23]: In our ACSL sources, their is no definition of these operations but we could suppose that ACSL rely on the C standard definition.

WP [Bla+24; Bla20; Bau+24]: In WP, the `load` and `store` operations are defined according to the memory model. The memory is represented by one or several memory maps M depending on the value type for the Typed memory model. A `load` at the address l is modeled by $M[l]$. A `store` of the value v at the address l is modeled by $M[l \leftarrow v]$. These operations can be restricted for some types or change depending of the memory model. For examples, these operations on pointers can not be performed in the Hoare memory model and in the Byte memory model, they are translated into manipulation of ranges of bits. The `store` operation modifies the memory but not the `store` operation. We suppose the type of these operations could be defined as:

$$\texttt{load}\colon \text{mem} \times \text{addr} \to \text{val}$$
$$\texttt{store}\colon \text{mem} \times \text{addr} \times \text{val} \to \text{mem}$$

Region analysis [Dam+25]: In [Dam+25], 4 memory operations are considered: `load`, `store` and two `shift` operations (one on indices and one on structure or union fields).

The `load` operation gets a value from the memory at a specific address. It also checks the size of the data type and if the address is valid (i.e. allocated). The type of this operation could be defined as:

$$\texttt{load:}\ \ \texttt{mem} \times \texttt{addr} \times \texttt{type} \to \texttt{option val}.$$

The `store` operation writes a value in the memory at a specific address. It returns the updated memory if the operation succeeded. The type of this operation could be defined as:

$$\texttt{store:}\ \ \texttt{mem} \times \texttt{addr} \times \texttt{val} \to \texttt{option mem}.$$

These operations do not consider offset and address shifts. These are devolved to the shift operations. The shift operation for indexes (for array) computes a new address from an address and an offset. It also checks if the shift is valid depending on the size of the data type and the memory space. The type of this operation could be defined as:

$$\texttt{shift: mem} \times \texttt{addr} \times \mathbb{Z} \times \texttt{type} \rightarrow \texttt{option addr}.$$

The shift operation for fields (of structures or unions) computes a new address corresponding to the field of the structure or the union from the given address. We may assume that this operation could also have condition of succeed. The type of this operation could be defined as:

$$\texttt{shift: mem} \times \texttt{addr} \times \texttt{field} \rightarrow \texttt{option addr}.$$

2.7 Block/memory deallocation

C standard The C standard defines the `free` function to deallocate the memory space pointed by a pointer. Then this memory space can be re-allocated. This function has a different behavior if the pointer is null or if the memory space had already been deallocated. For the former, nothing occurs. The latter is an undefined behavior.

CompCert CompCert defines the `free` memory operation to deallocate a block in the memory as "$\texttt{free: mem} \times \texttt{block} \rightarrow \texttt{option mem}$" [LB08]. This operation fails if the block was already deallocated. Freeing a block as no side effect on other bocks. In the abstract model, this operation is defined as "$\texttt{free}\ m\ b := m[b := \bot]$" [Kre14]. In the concrete model, this operation is defined as "$\texttt{free}(m, b) =$
if not $m \models b$ then ε else $\lfloor N, B\{b \leftarrow [0,0)\}, F\{b \leftarrow true\}, C \rfloor$" [LB08] with (N, B, F, C) representing memory states. "Memory states (type `mem`) are quadruples (N, B, F, C), where

- N : `block` is the first block not yet allocated;
- B : $\texttt{block} \rightarrow \mathbb{Z} \times \mathbb{Z}$ associates bounds to each block reference;
- F : $\texttt{block} \rightarrow \texttt{boolean}$ says, for each block, whether it has been deallocated (`true`) or not (`false`);
- C : $\texttt{block} \rightarrow \mathbb{Z} \rightarrow \texttt{option}$ (`memtype` $\times$ `val`) associates a content to each block b and each byte offset i. A content is either ε, meaning "invalid", or $\lfloor \tau, v \rfloor$, meaning that a value v was stored at this location with type τ." [LB08]

This operation means "**Freeing** a block simply sets its deallocated status to true, rendering this block invalid, and its bounds to [0, 0), reflecting the fact that this block no longer occupies any memory space." [LB08]

In version 2 of CompCert [Ler+12], this operation is redefined as:

$$\texttt{free: mem} \to \texttt{block} \to \mathbb{Z} \to \mathbb{Z} \to \texttt{option mem}$$

The operation `free` $m\,b\,l\,h$ frees the range of offsets $[l, h)$ (the bound h is not included) of the block b. The previous version of this operation freed the whole block. Version 2 mentions "Another change is that `free` can now fail, typically if the locations to be freed have been freed already", but this was already the case in the previous version.

The `free` operation from CompCert does not exactly correspond to the `free` function in the C standard, it models "destruction" of blocks "at function return" [Ler+14] and "could also be used to model acquisition of address space via system calls" [Ler+14].

Krebbers Krebbers considers that deallocated addresses stay well-typed but forbids their use in pointer arithmetic. Krebbers defines the `free` memory operation to deallocate an object as "`free : mem` $\to$ `index` $\to$ `mem`" [Kre13]. Its definition is "`free` $m\ x$ `:= ` m`[`x `:= freed (indextype`$_m$ x`)]`" [Kre13]. This means the memory is updated with the value `freed` (indextype$_m$ x) for x, The function `indextype`$_m$ returns the type of the variable x. In his thesis, Krebbers defines this operation differently:

$$\texttt{free : index} \to \texttt{mem} \to \texttt{mem}$$
$$\texttt{free}\ o\ m := m\texttt{[}o\texttt{:=typeof}\ w\texttt{]} \text{ if } m\ o = (w, \mu)$$

"The operation `free` o m deallocates the object o in m, and keeps track of the type of the deallocated object. These two definitions seems to be equivalent, `indextype` and `typeof` returning the type of the variable, and the keyword `freed` does not seem to have an impact on this operation. In order to deallocate dynamically obtained memory via `free`, the side-condition `freeable` a m describes that the permissions are sufficient for deallocation, and that a points to the first element of a `malloced` array." [Kre15]

$$\texttt{freeable : addr} \to \texttt{mem} \to \texttt{Prop}$$
$$\texttt{freeable}\ a\ m := \exists\ o\ \tau\ \sigma\ n\ w\ .\ a = (o : \tau, \xrightarrow{\tau[n]} 0, 0)_{\tau > *\sigma}, mo = (w, \text{true})$$

and all $\bar{w}$ have the permission $\Diamond(0, 1)$

ACSL In ACSL, it is possible to specify a behavior about deallocation, with the clause `frees`. If it is not used, it means that there is no newly deallocated memory block. The predicate `\freeable` allow to verify if a pointer can be safely released using the C function free. A difference between ACSL and the C standard : the null pointer is considered as not freeable for ACSL however it is a valid argument to the C function free [Bau+23].

2.8 Casts

The cast operation corresponds to converting a variable from a type to another type. Therefore, in some sources, the term "conversion" (and similar words) can be found.

C standard: The C standard defines implicit arithmetic conversions for operators that expect arithmetic type operands, but explicit casts are allowed too. It also defines specific rules when casting integers to floating types (or the other way) to consider carefully the fractional part. The C standard specifies many rules when casting involves pointer types, according various cases. For example, "the left operand has atomic, qualified, or unqualified arithmetic type, and the right has arithmetic type" [ISO11, §6.5.16.1]. However, it prohibits casting between a pointer type and a floating-point type, unlike the casting between a pointer type and an integer, where they can be converted from one to the other. Even when converting an integer to a pointer type is possible, the result of this operation is "is implementation-defined" [ISO11, §6.3.2.3] due to alignment issues and potential trap representations.

CompCert: Even if the CompCert semantics is "untyped" [KLW14], CompCert guarantees that the result of a cast is well-typed. CompCert defines a function to perform cast: `convert:val` $\times$ `memtype` $\rightarrow$ `val`. This function is defined for some type of `val` (for `int`, `float`, pointers and uninitialized values) and for some `memtype` (for signed and unsigned `int` and `float`, for 8-bit int to 64-bit float).

Krebbers: It is not explained in our sources for Krebbers [Kre13] how casts are considered. However some information related to casts are provided: integer promotions and integer demotions require explicit cast. Krebbers defines specific functions to covert values from and to bits.

Krebbers does not have a specific operation for casting variables. But he defines a cast invariant that each "address" (i.e., a pointer) should satisfy: $\tau >_* \sigma_p$. This invariant signifies "type τ is pointer castable to σ_p" [Kre16]. It is "inductively defined by $\tau >_* \tau$, $\tau >_*$ `unsigned char`, and $\tau >_*$ `any`" [Kre16]. The `any` type is a specific type from Krebbers to represent the C standard type `void *`. The respect for this invariant is determined at run time.

ACSL: ACSL performs some conversions similarly to the C standard, like casting "from a C integer type or a float type to a float or a double" [Bau+23], but ACSL has different casting rules than C standard. The ones found in our ACSL sources are summarized in Table 1. ACSL performs some implicit casts, like float in real, but unlike in the C standard,

some casts have to be explicit. For example, "there is no implicit cast from an array type to a pointer type" [Bau+23]. This implies that "`t[i]` is not always equivalent to `*(t+i)`" [Bau+23]. On the contrary, some casts are implicit with ACSL but not in the C standard. For example, "from an array of a given size to an array with unspecified size" [Bau+23]. In ACSL there are differences when casting C variables or logical variables. For example, casting an array type to a pointer type is forbidden for logical variables.

Table 1: Different casting rules between the C Standard and ACSL

CASTING RULE	**C**	**ACSL**
Implicit cast from an array type to a pointer type	✓	×
Implicit cast from an array of a given size to an array with unspecified size	×	✓
Allowed cast between structure types	×	✓
Explicit cast between aggregate types	×	✓

WP: Heterogeneous casts cannot be represented in the WP memory models but "variants of the Typed model enable limited forms of casts" [Bau+24]. However it is not explained which ones and what are these limitations.

Region analysis: In the region analysis of WP, access to a union field is similar to a cast operation (the memory region is not changed) [Dam+25]. It also mentions it is necessary to keep a byte representation of the value for conversion between types. It is necessary when accessing a variable with a different type than the type used during its definition.

3 Properties

3.1 Pointer properties

3.1.1 Pointer equality

C standard The C standard defines cases where two pointers are equal :

- if both are null pointers ;
- if both point to the same object or function ;
- if both point to one past the last element of the same array object ;
- if both point to members of the same union object ;

– if "one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space".

The latter is due to rules about objects adjacent in memory. If a previous pointer operation, on this pointer, conducted to an undefined behavior then the equality operation also produces an undefined behavior.

CompCert In CompCert if two pointers have disjoint block identifiers then they cannot be equal.

ACSL ACSL allows equality operation on invalid pointers, and two invalid pointers can be equal. These pointers can be invalid for reading or valid for reading and even not initialized. ACSL also states that the equality operation is reflexive, even on pointers not initialized or not valid for reading.

WP In WP, when two pointers alias, they are equal. In our sources, it is not mentioned if two invalid pointers may be equal.

3.1.2 Pointer comparison

C standard "When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to." Some comparison results on pointers are defined in the C standard. The rules for equal pointers are presented in section 3.1.1 but the C standard also defines rules for pointer comparison. Consider two pointers, if both point to:

– members of the same aggregate object then the pointer to the structure member declared later compares **greater than** the one declared earlier;

– elements of the same array then the pointer with a larger subscript compares **greater than** the one with a lower subscript.

As for equality operation, if a previous pointer operation conducted to an undefined behavior then the comparison operations also produce undefined behaviors.

CompCert CompCert allows pointer comparison depending on their validity status. For its version 1.12 (and previous versions), CompCert requires that both pointers are valid. For its next versions, CompCert distinguishes two cases: when the pointers are in the same block or not. If the pointers are in different blocks, they have to be *valid*. However, if they are in the same block, they just need to be *weakly valid* (definition provided in Section 3.1.4).

ACSL ACSL provides a predicate, `\pointer_comparable`, to check if two pointers are comparable. It only holds when both pointers point to "the same function or to an element (or one past the last element) of the same array object" [Bau+23]. In our sources for ACSL, it is specified that "comparing two pointers may also lead to undefined behaviors" [Bau+23]. They do not precise in which cases but we can assume it is based on the C standard.

3.1.3 Pointer separation

CompCert In CompCert, it is "guaranteed that the memory areas corresponding to two distinct variables or two successive calls to `malloc` are disjoint" [Ler+14]. Moreover, in CompCert, " blocks are separated by construction" [Ler+14]. This implies, pointer are separated by construction and pointer arithmetic cannot not modify this separation. Either the pointer arithmetic create a valid pointer respecting this separation property or create an invalid pointer.

Krebbers Krebbers defines a specific operator, $\perp$, to denotes that two addresses are disjoint. When two addresses are disjoint, this means that "somewhere along their path from the top of the whole object to their subobject they take a different branch at an array of struct subobject" [Kre13]. This definition is very different than the definition of pointer separation in other models. This is due to the representation of pointers that is also very different than in the others.

ACSL ACSL provides a predicate, `\separated`, on a list of memory locations expressing that these memory locations are separated from each other. This predicate is defined as [Bau+23]:

$$
\begin{aligned}
\backslash separated(s_1, s_2) \quad \triangleq \quad & \forall p \in s_1, q \in s_2, \\
& \forall \text{ integer } i, j, \\
& 0 \leq i < \texttt{sizeof}(*p), \\
& 0 \leq j < \texttt{sizeof}(*q) \Rightarrow \\
& \texttt{(char*)p + i != (char*)q + j}
\end{aligned}
$$

Our ACSL sources also state that two pointers are different if "either their base is different or their offset is different" [BMP24].

WP WP relies on the predicate `\separated` of ACSL to specify when pointers are separated. Besides, WP also considers that "locals, formals and global variables are indeed separated from each others" [Bla+24] and "pointers with different bases necessarily point to disjoint memory locations." [Bla+24]. In WP, by definition, variable addresses have unique base, and therefore are separated. Moreover the Ref memory model of WP as-

sumes that pointer accessed with dereferencing are separated from any other pointers.

3.1.4 Pointer validity

CompCert CompCert provides a definition of pointer validity: "A pointer is valid if its offset is strictly within the block bounds" [KLW14]. Besides CompCert defines a less strict validity status, *weakly valid*: "A pointer is weakly valid if it is valid or end-of-array." [KLW14]. This status is used for pointer comparison.

Krebbers Krebbers formalizes *well-formed* environment. This notion includes *valid types*, and *valid base types* (a pointer is a "base type"). The *valid base types* for pointers rely on a formalization of "point-to types τ_p to which a pointer may point". "The judgment $\Gamma \vdash_* \tau_p$ describes *point-to types τ_p to which a pointer may point*"[Kre16]:

$$\frac{}{\Gamma \vdash_* \texttt{any}} \qquad \frac{\Gamma \vdash_* \vec{\tau} \qquad \Gamma \vdash_* \tau}{\Gamma \vdash_* \vec{\tau} \to \tau} \qquad \frac{\Gamma \vdash_b \tau_b}{\Gamma \vdash_* \tau_b} \qquad \frac{\Gamma \vdash \tau \qquad n \neq 0}{\Gamma \vdash_* \tau[n]}$$

$$\frac{}{\Gamma \vdash_* \texttt{struct } t} \qquad \frac{}{\Gamma \vdash_* \texttt{union } t}$$

ACSL ACSL provides several predicates about the validity of a pointer: `\valid`, `\valid_read` because it is possible "in C to create entirely invalid pointers" [BMP24]. `\valid` "expresses that a memory location can be safely read or written" [BMP24]. It holds if the pointer "is guaranteed to produce a definite value according to the C standard" [BMP24]. `\valid_read` expresses that the memory can be safely read but might not be writable. This implies that if `\valid` on a pointer holds then `\valid_read` also holds. It is not defined in our sources for ACSL when a pointer can be safely read or not, but we can assume it follows the C standard definition. However, they provide an example: "it is allowed to read from a string literal, but not to write into it" [Bau+23]. Our ACSL sources warn about another operation, "adding or subtracting an offset to a pointer [...] may also lead to undefined behaviors" [Bau+23]. ACSL states the null pointer is always invalid, even for reading. These predicates hold depending on the type of their argument. For example: "`\valid{L}((int *) p)` and `\valid{L}((char *)p)` are not equivalent" [Bau+23].

WP Wp relies on the predicate `\valid` provided by ACSL to assume the validity of pointers. However, the memory model *Ref* of WP infers validity and separation hypotheses of pointers.

Region analysis The Region analysis of WP relies on a *coherent memory* property to determine the validity of a pointer. The *coherent memory* property is defined as following: if an address a_1 points to an address a_2, the region r_1 points to the region r_2 and a_1 belongs to r_1 then a_2 belongs to r_2.

3.2 Memory properties

3.2.1 Block validity

CompCert Instead of pointer validity, CompCert defines a notion of *block reference* validity. A "block reference is **valid** if it was previously allocated but not yet deallocated" [LB08]. This means that allocation operation (`alloc`) and deallocation operation (`free`) impact the validity of a block. Allocation operation returns a valid block that is *fresh*, i.e. distinct from the other valid blocks in the memory. There is a difference for the validity of a deallocated block in CompCert, depending on the memory model. In the concrete model, a deallocated block is invalid (and empty) but in the abstract model, its validity status is unspecified. In CompCert, the `store` operation has no side effects on the validity of a block, its validity is preserved.

In version 2, CompCert does not rely anymore on the bounds of the block, but uses the permissions. A location (b, i) in the block b, at the index i, is valid if the permission associated with this location is `Nonempty`.

3.2.2 Bound information

CompCert In CompCert, blocks have specified bounds. Its concrete and abstract models, both have a function that associates bounds to a block reference. These bounds are given to the allocating operation when allocation a new block. Allocation, deallocation and store operations on a block do not have side effects on the bounds of other separated blocks, their bounds are preserved.

3.2.3 Valid access

CompCert To determine the validity of a memory access, CompCert relies on the *Block validity*, the *bound information* and the type of the writing. The *valid access* is a relation that states if it is valid to write into a memory location. The load and store operations succeed if and only if the corresponding memory reference is valid. In other words, if the access is valid then the load or store operations succeed and if they succeed then the access is valid.

3.2.4 Block freshness

CompCert CompCert defines a specific relation to represent the *freshness* of a block. A block is fresh when "distinct from any other block that was valid in the initial memory state" [LB08]. Only the allocation operation has an effect on the freshness of a block. Deallocaation, load or store operation have no effects on the freshness of blocks.

3.2.5 Strict aliasing

Krebbers Krebbers provides a formal definition of the *Strict-aliasing* property as a theorem. This theorem states : Let two frozen addresses a_1 and a_2 that have the types σ_1 and σ_2 in a valid memory, where σ_1 and σ_2 are not unsigned char type, and σ_1 and σ_2 are not subtype of each other. A frozen addresses are pointers stored in memory or used as arguments of a function. They can not be used for type-punning. Therefore a_1 and a_2 are disjoint or "accessing a_1 after accessing a_2 and vice versa fails" [Kre13].

Krebbers formalizes a theorem for *Strict-aliasing*: "*Given* $\Gamma, \Delta \vdash m$, *frozen addresses* a_1 *and* a_2 *with* $\Delta, m \vdash a_1 : \sigma_1$ *and* $\Delta, m \vdash a_2 : \sigma_2$ *and* $\sigma_1, \sigma_2 \neq$ `unsigned char`, *then either:*

1. *Whe have* $\sigma_1 \subseteq_\Gamma \sigma_2$ *or* $\sigma_2 \subseteq_\Gamma \sigma_1$.
2. *Whe have* $a_1 \perp_\Gamma a_2$.
3. *Accessing* a_1 *after accessing* a_2 *and vice versa fails. That means:*
   (a) $(\texttt{force}_\Gamma\ a_2\ m)\langle a_1\rangle_\Gamma = \perp$ *and* $(\texttt{force}_\Gamma\ a_1\ m)\langle a_2\rangle_\Gamma = \perp$, *and*
   (b) $m\langle a_2 := v_1\rangle_\Gamma\langle a_1\rangle_\Gamma = \perp$ *and* $m\langle a_1 := v_2\rangle_\Gamma\langle a_2\rangle_\Gamma = \perp$ *for all stored values* v_1 *and* v_2." *[Kre16]*

WP The Typed memory model of WP "enforces strict aliasing", by forbidding some casts and union reading, but our sources of WP do not define the *strict aliasing*.

3.3 Cast properties

3.3.1 Type punning

C standard The C standard provides a definition of *type punning*: "If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type" [ISO11]. It also warns this might conduct to a trap representation.

Krebbers Krebbers states *type punning* correspond to "a union can be read using a pointer to another variant than the current one" [Kre13]. But for Krebbers, *type punning* is allowed only in certain conditions. To identify these conditions, it relies on the GCC documentation. To ensure *type punning*, Krebbers uses an annotation on pointer definition. So, when some operations are performed with the pointer, this annotation change and then this pointer can not be used for *type punning* anymore. The considered operations are: store a pointer in the memory and use the pointer as an argument of a function.

References

[Bau+23] Patrick Baudin et al. *ANSI/ISO C Specification Language*. Version 1.20. 2023. URL: `https://www.frama-c.com/download/acsl.pdf`.

[Bau+24] Patrick Baudin et al. *WP plug-in manual - For Frama-C 29.0 (Copper)*. 2024. URL: `https://www.frama-c.com/download/frama-c-wp-manual.pdf`.

[Bla+24] Allan Blanchard et al. "Formally Verifying That a Program Does What It Should: The Wp Plug-in". In: *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*. Ed. by Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. Cham: Springer International Publishing, 2024, pp. 187–261. DOI: `10.1007/978-3-031-55608-1_4`.

[Bla20] Allan Blanchard. *Introduction to C program proof with Frama-C and its WP plugin*. 2020. URL: `https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf`.

[BMP24] Allan Blanchard, Claude Marché, and Virgile Prevosto. "Formally Expressing What a Program Should Do: The ACSL Language". In: *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*. Ed. by Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles. Cham: Springer International Publishing, 2024, pp. 3–80. DOI: `10.1007/978-3-031-55608-1_1`.

[Dam+25] Jérémy Damour et al. "Formalisation d'une analyse de région pour Frama-C/WP". In: *36es Journées Francophones des Langages Applicatifs (JFLA 2025)*. 2025. URL: `https://hal.science/hal-04859489v1/file/jfla2025-final68.pdf`.

[ISO11] ISO Central Secretary.
Information technology – Programming languages – C.
Standard ISO/IEC 9899:2011. Geneva, Switzerland:
International Organization for Standardization, 2011.

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk.
"Formal C Semantics: CompCert and the C Standard".
In: *Interactive Theorem Proving.*
Ed. by Gerwin Klein and Ruben Gamboa.
Cham: Springer International Publishing, 2014, pp. 543–548.
DOI: 10.1007/978-3-319-08970-6_36.

[Kre13] Robbert Krebbers.
"Aliasing Restrictions of C11 Formalized in Coq".
In: *Certified Programs and Proofs.*
Ed. by Georges Gonthier and Michael Norrish.
Cham: Springer International Publishing, 2013, pp. 50–65.
DOI: 10.1007/978-3-319-03545-1_4.

[Kre14] Robbert Krebbers. "An operational and axiomatic semantics for non-determinism and sequence points in C".
In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*
POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 101–112.
DOI: 10.1145/2535838.2535878.

[Kre15] Robbert Krebbers. "The C standard formalized in Coq".
Radboud University Nijmegen, 2015.
URL: https://robbertkrebbers.nl/research/thesis.pdf.

[Kre16] Robbert Krebbers.
"A Formal C Memory Model for Separation Logic".
In: *Journal of Automated Reasoning* (2016).
DOI: 10.1007/s10817-016-9369-1.

[KW13] Robbert Krebbers and Freek Wiedijk. "Separation logic for non-local control flow and block scope variables".
In: *Proceedings of the 16th International Conference on Foundations of Software Science and Computation Structures.*
FOSSACS'13. Rome, Italy: Springer-Verlag, 2013,
pp. 257–272. DOI: 10.1007/978-3-642-37075-5_17.

[LB08] Xavier Leroy and Sandrine Blazy.
"Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations". In: *Journal of Automated Reasoning* 41.1 (July 2008), pp. 1–31.
ISSN: 1573-0670. DOI: 10.1007/s10817-008-9099-0.

[Ler+12] Xavier Leroy et al. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA, 2012, p. 26. URL: https://inria.hal.science/hal-00703441/document.

[Ler+14] Xavier Leroy et al. "Program Logics for Certified Compilers". In: ed. by Andrew Appel. Cambridge University Press, 2014. Chap. The CompCert memory model, pp. 237–271. DOI: 10.1017/CBO9781107256552.037.