Aliasing restrictions of C11 formalized in Coq

Robbert Krebbers

ICIS, Radboud University Nijmegen, The Netherlands

Abstract. The C11 standard of the C programming language describes dynamic typing restrictions on memory operations to make more effective optimizations based on alias analysis possible. These restrictions are subtle due to the low-level nature of C, and have not been treated in a formal semantics before. We present an executable formal memory model for C that incorporates these restrictions, and at the same time describes required low-level operations.
Our memory model and essential properties of it have been fully formalized using the Coq proof assistant.

1 Introduction

Aliasing is when multiple pointers refer to the same object in memory. Consider:

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

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

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

In *type-based alias analysis*, type information is used to determine whether pointers are aliased or not. Given the following example

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

a compiler should be able to assume that `p` and `q` are not aliased as their types differ. However, the static type system of C is too weak to enforce this restriction because a union type can be used to call `g` with aliased pointers.

```
union { int x; float y; } u = { .y = 3.14 }; g(&u.x, &u.y);
```

A union is C's version of a *sum* type, but contrary to ordinary sum types, unions are *untagged* instead of *tagged*. This means that their current variant cannot be obtained. Unions destroy the property that each memory area has a unique type that is statically known. The *effective type* [6, 6.5p6-7] of a memory area thus depends on the *run time behavior* of the program.

The *strict-aliasing restrictions* [6, 6.5p6-7] imply that a pointer to a variant of a union type (not to the whole union itself) can only be used for an access (a read

or store) if the union is in that particular variant. Calling `g` with aliased pointers (as in the example where `u` is in the `y` variant, and is accessed through a pointer `p` to the `x` variant) thus results in *undefined behavior*, meaning the program may do literally anything. C uses a "garbage in, garbage out" principle for undefined behavior to refrain compilers from having to insert (possibly expensive) checks to handle corner cases. A compiler thus does not have to generate code that tests whether effective types are violated (here: to test whether `p` and `q` are aliased), but is allowed to assume no such violations occur.

As widely used compilers (*e.g.* GCC and Clang) perform optimizations based on C's aliasing restrictions, it is essential to capture these in a formal memory model for C. Not doing so, makes it possible to prove certain programs to be correct when they may crash when compiled with an actual C compiler.

Approach. The main challenge of formalizing C's strict-aliasing restrictions is that both *high-level* (by means of typed expressions) and *low-level* (by means of byte-wise manipulation) access to memory is allowed. Hence, an abstract "Java-like" memory model would not be satisfactory as it would disallow most forms of byte-wise manipulation.

Significant existing formal semantics for C (*e.g.* Leroy *et al.* [10], Ellison and Rosu [3]) model the memory using a finite partial function to objects, where each object consist of an array of bytes. Bytes are symbolic to capture indeterminate storage and pointer representations. However, because no information about the variants of unions is stored, this approach cannot capture C's strict-aliasing restrictions. We refine this approach in two ways.

- Instead of using an array of bytes as the contents of each object, we use well-typed trees with arrays of bits that represent base values as leafs.
- We use symbolic bits instead of bytes as the smallest available unit.

The first refinement is to capture strict-aliasing restrictions: effective types are modeled by the state of the trees in the memory model. Our use of trees also captures restrictions on padding bytes[1] simply because these are not represented. The second is to deal with bit fields as part of structs (in future work) where specific bits instead of whole bytes may be indeterminate.

The novelty of our memory model is that it also describes low-level operations such as byte-wise copying of objects and type-punning. As depicted in Figure 1, the model has three layers: (a) *abstract values*: trees with mathematical integers and pointers as leafs, (b) *memory values*: trees with arrays of bits as leafs, and (c) arrays of bits. Memory values are internal to the memory model, and abstract values are used for its external interface. Pointers are represented by a pair of a cell identifier and a path through the corresponding memory value.

In order to enable type-based alias analysis, we have to ensure that only under certain conditions a union can be read using a pointer to another variant than

[1] In particular: "When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values" [6, 6.2.6.1p6].

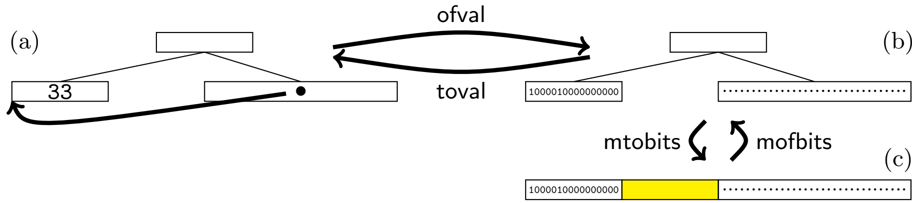

Fig. 1. The representations of `struct { short x, *p; } s = { 33; &s.x }`.

the current one (this is called *type-punning* [6, 6.5.2.3]). Since the C11 standard is unclear about these conditions[2], we follow the GCC documentation [4] on it. It states that “type-punning is allowed, provided the memory is accessed through the union type”. This means that the function `f` has defined behavior[3]:

```
union U { int x; float y; };
int f() { union U t; t.y = 3.0; return t.x; }
```

whereas the function `g` exhibits undefined behavior:

```
int g() { union U t; int *p = &t.x; t.y = 3.0; return *p; }
```

We formalize the previously described behavior by decorating the formal definition of pointers with annotations. Whenever a pointer to a variant of some union is stored in memory, or used as the argument of a function, the annotations are changed to ensure that type-punning is no longer possible via that pointer.

We tried to follow the C11 standard [6] as closely as possible. Unfortunately, it is often ambiguous due to its use of natural language (see the example above, this message [12] on the standard’s committee’s mailing list, and Defect Report #260 and #236 [5]). In the case of ambiguities, we tried to err on the side of caution. Generally, this means assigning undefined behavior.

Related work. The first formalization of a significant part of C is due to Norrish [14] using HOL4. He considered C89, in which C’s aliasing restrictions were not introduced yet, and thus used a memory model based on just arrays of bytes. Tuch *et al.* [18] also consider a memory model based on just arrays of bytes.

Leroy *et al.* have formalized a large part of a C memory model as part of CompCert; a verified optimizing C compiler in Coq [11,10]. The first version of their memory model [11] uses type-annotated symbolic bytes to represent integer, floating point, and pointer values. This version describes some aliasing restrictions (namely those on the level of base types), but at the cost of prohibiting any kind of “bit twiddling”. In the second version of their memory model [10], type information has been removed, and symbolic bytes were only used for pointer values and indeterminate storage. Integers and floating points were represented

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

[3] Provided `size_of(int)` $\leq$ `size_of(float)` and `int`s do not have trap values.

using numeric bytes. We adapt their choice of using symbolic representations for indeterminate storage and pointers. Moreover, we adapt their notion of *memory extensions* [11]. As an extension of CompCert, Robert and Leroy have verified an untyped alias analysis [16].

Ellison and Rosu [3] have defined an executable semantics of the C11 standard in the $\mathbb{K}$-framework. Their memory model is based on the CompCert memory model by Leroy *et al.* and does not describe the aliasing restrictions we consider.

The idea of a memory model that uses trees instead of arrays of plain bits, and paths instead of offsets to model pointers, has been used for object oriented languages before. It goes back to at least Rossie and Friedman [17], and has been used by Ramananandro *et al.* [15] for C++. However, we found no evidence in the literature of using trees to define a memory model for C.

Contribution. This work presents an executable mathematically precise version of a large part of the (non-concurrent) C memory model. In particular:

- We give a formal definition of the core of the C type system (Section 2).
- Our formalization is parametrized by an abstract interface to allow implementations that use multiple integer representations (Section 3).
- We define a memory model that describes a large set of subtly interacting features: effective types, byte-level operations, type-punning, indeterminate memory, and pointers "one past the last element" (Sections 4 to 6).
- We demonstrate that our memory model is suitable for formal proofs by verifying essential algebraic laws, an abstract version of `memcpy`, and an essential property for aliasing analysis (Section 6).
- All proofs have been formalized using the Coq proof assistant (Section 7).

As this paper describes a large formalization effort, we often just give representative parts of definitions due to space restrictions. The interested reader can find all details online as part of our Coq formalization.

Notations. We let B^{opt} denote the *option type*, which is inductively defined as either $\bot$ or x for some $x \in B$. We often implicitly lift operations to operate on the option type, which is done using the *option monad* in the Coq formalization. A *partial function* $f : A \to B^{\mathsf{opt}}$ is called *finite* if its *domain* $\mathsf{dom}\ f$ is finite. The operation $f[x := y]$ stores the value y at index x.

2 Types

We treat the most relevant C-types: integers, pointers, arrays, structs, unions, and the void type. Floating point and function types are omitted as these are orthogonal to the aliasing restrictions described in this paper. The void type plays a dual role, it is used for functions without a return value, and for pointers to data of an unspecified type.

Definition 2.1. Integer*,* base*, and* full types *are inductively defined as:*

$$\begin{aligned} si \in \mathsf{signedness} &::= \mathsf{signed} \mid \mathsf{unsigned} \\ \tau_\mathsf{i} \in \mathsf{inttype} &::= si\ k \\ \tau_\mathsf{b} \in \mathsf{basetype} &::= \tau_\mathsf{i} \mid \tau{*} \\ \tau \in \mathsf{type} &::= \tau_\mathsf{b} \mid \mathsf{void} \mid \tau[n] \mid \mathsf{struct}\ s \mid \mathsf{union}\ u \end{aligned}$$

In the above definition, k ranges over *integer ranks* (see Section 3), and $s, u \in \mathsf{tag}$ range over struct and union names (called *tags*). *Environments* ($\Gamma \in \mathsf{env}$) are finite partial functions from tags to lists of types representing struct and union fields. Since fields are represented using lists, they are nameless. We allow structs and unions with the same name for simplicity.

The above definition still allows ill-formed types as `void[0]`. Also, we have to ensure that cyclic structs and unions are only allowed when recursion is guarded by a pointer. The type `struct T1 { struct T1 x; };` should thus be prohibited whereas `struct T2 { struct T2 *p; };` should be allowed.

Definition 2.2. *The judgment $\Gamma \vdash_\mathsf{b} \tau_\mathsf{b}$ describes valid base types, $\Gamma \vdash \tau$ valid types, and $\Gamma \vdash_* \tau$ types to which pointers are allowed.*

$$\frac{}{\Gamma \vdash_\mathsf{b} \tau_\mathsf{i}} \quad \frac{\Gamma \vdash_* \tau}{\Gamma \vdash_\mathsf{b} \tau{*}} \qquad \frac{\Gamma \vdash_\mathsf{b} \tau_\mathsf{b}}{\Gamma \vdash \tau_\mathsf{b}} \quad \frac{\Gamma \vdash \tau \quad 0 < n}{\Gamma \vdash \tau[n]} \quad \frac{\Gamma\, s = \vec{\tau}}{\Gamma \vdash \mathsf{struct}\ s} \quad \frac{\Gamma\, u = \vec{\tau}}{\Gamma \vdash \mathsf{union}\ u}$$

$$\frac{\Gamma \vdash_\mathsf{b} \tau_\mathsf{b}}{\Gamma \vdash_* \tau_\mathsf{b}} \quad \frac{}{\Gamma \vdash_* \mathsf{void}} \quad \frac{\Gamma \vdash \tau \quad 0 < n}{\Gamma \vdash_* \tau[n]} \quad \frac{}{\Gamma \vdash_* \mathsf{struct}\ s} \quad \frac{}{\Gamma \vdash_* \mathsf{union}\ u}$$

The judgment for well-formed environments Γ valid is defined as:

$$\frac{}{\emptyset\ \mathsf{valid}} \qquad \frac{\Gamma\ \mathsf{valid} \qquad \Gamma \vdash \vec{\tau} \qquad s \notin \mathsf{dom}\ \Gamma \qquad 0 < |\vec{\tau}|}{(s : \vec{\tau},\, \Gamma)\ \mathsf{valid}}$$

Due to the fact that C allows (mutually) recursive struct and union types, we allow pointers to struct and union types before they are declared in the $\Gamma \vdash_* \tau$ judgment. Note that $\Gamma \vdash \tau$ does not imply Γ valid.

Well-formedness of $\Gamma = \mathtt{T2} : [\,\mathsf{struct}\ \mathtt{T2}{*}\,]$ can be derived using the judgments $\emptyset \vdash_* \mathsf{struct}\ \mathtt{T2}$, $\emptyset \vdash_\mathsf{b} \mathsf{struct}\ \mathtt{T2}{*}$, $\emptyset \vdash \mathsf{struct}\ \mathtt{T2}{*}$, and thus Γ valid. The environment $\mathtt{T1} : [\,\mathsf{struct}\ \mathtt{T1}\,]$ is ill-formed because we do not have $\emptyset \vdash \mathsf{struct}\ \mathtt{T1}$.

Lemma 2.3. *Given an arbitrary set A, and functions $f_\mathsf{b} : \mathsf{basetype} \to A$, $f_\mathsf{a} : \mathsf{type} \to \mathbb{N} \to A \to A$, $f_\mathsf{s}, f_\mathsf{u} : \mathsf{tag} \to \mathsf{list\ type} \to \mathsf{list}\ A \to A$, the function $\mathsf{type_iter} : \mathsf{env} \to \mathsf{type} \to A$ is total for well-formed environments and types.*

$$\begin{aligned} \mathsf{type_iter}_\Gamma\ \tau_\mathsf{b} &:= f_\mathsf{b}\ \tau_\mathsf{b} \\ \mathsf{type_iter}_\Gamma\ (\tau[n]) &:= f_\mathsf{a}\ \tau\ n\ (\mathsf{type_iter}_\Gamma\ \tau) \\ \mathsf{type_iter}_\Gamma\ (\mathsf{struct}\ s) &:= f_\mathsf{s}\ s\ (\Gamma\, s)\ (\mathsf{type_iter}_\Gamma\ (\Gamma\, s)) \\ \mathsf{type_iter}_\Gamma\ (\mathsf{union}\ u) &:= f_\mathsf{u}\ u\ (\Gamma\, u)\ (\mathsf{type_iter}_\Gamma\ (\Gamma\, u)) \end{aligned}$$

We often lift $\mathsf{type_iter}\ \Gamma$ to operate pointwise on lists of types.

The previous lemma is used to define functions where recursion on fields of unions and structs is needed. Totality is proven by well-founded induction on the size of the type environment.

Our formalization of the C type system differs in various ways from existing work. In CompCert [10], fields of structs and unions are not stored in an environment, but are stored in the types itself. Hence, instead of having a construct $\mathsf{struct}\ s$, they have a construct $\mathsf{struct}\ s\ \vec{\tau}$ and a special pointer type $\mathsf{struct_ptr}\ s$ to allow recursive structs. Although this relieves one from having to carry a type environment around, the main disadvantage is that one has to roll and unroll types at certain places, and that one loses canonicity.

Affeldt and Marti [1] have also formalized a part of the C type system. Like us, they use an environment to capture the types of fields of structs, but they define non-cyclicity of type environments using a complex constraint on paths through types. Our definition $\Gamma\ \mathsf{valid}$ follows the structure of type environments, and seems more easy to use (for example for proving termination of the iteration function type_iter). Also, they omit union types, and do not parametrize by an abstract interface to allow multiple integer implementations.

3 Integer arithmetic

In order to make C portable, the C standard gives compilers a lot of freedom to represent integers and to perform integer arithmetic. First of all, it does not specify the sizes of integer types. For example, `signed int` does not necessarily have to be 32 bits, use two's complement representation, and be able to exactly hold values between -2^{31} and $2^{31}-1$. Only some minimum limits are described [6, 5.2.4.2.1]. Secondly, the standard puts few constraints on the way integers are represented as bits. Thirdly, overflow of signed integers is undefined behavior, whereas it wraps around modulo for the case of unsigned integers.

In order to capture different integer implementations, our memory model is parametrized by an abstract interface of *integer implementations*. This interface consists of a set K of *integer ranks* and functions:

$$\begin{aligned}
\mathsf{char} &: K & \mathsf{endianize} &: K \to \mathsf{list\ bool} \to \mathsf{list\ bool}\\
\mathsf{int} &: K & \mathsf{deendianize} &: K \to \mathsf{list\ bool} \to \mathsf{list\ bool}\\
\mathsf{ptr_rank} &: K & \mathsf{int_binop_ok} &: \mathsf{inttype} \to \mathsf{binop} \to \mathbb{Z} \to \mathbb{Z} \to \mathsf{bool}\\
\mathsf{char_bits} &: \mathbb{N}_{\geq 8} & \mathsf{int_binop} &: \mathsf{inttype} \to \mathsf{binop} \to \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}\\
\mathsf{rank_size} &: K \to \mathbb{N}_{>0} & \mathsf{int_cast_ok} &: \mathsf{inttype} \to \mathbb{Z} \to \mathsf{bool}\\
& & \mathsf{int_cast} &: \mathsf{inttype} \to \mathbb{Z} \to \mathbb{Z}
\end{aligned}$$

Here, binop is the inductive type of the C binary operations.

$$op \in \mathsf{binop} ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{<<} \mid \texttt{>>} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{==} \mid \texttt{<=} \mid \texttt{<} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}}$$

Unary operations are derived from the binary operations.

The rank char is the rank of the smallest available integer type, and ptr_rank the rank of the types `size_t` and `ptrdiff_t`. At an actual machine char corresponds to a byte, and its bit size is char_bits (called `CHAR_BIT` in the C header files). The function $\mathsf{rank_size}\ k$ gives the byte size of an integer with rank k.

Since all modern architectures use two's complement representation, we allow representations to differ solely in endianness. The function endianize takes a list of bits in little endian order and permutes them according to the implementation's endianness. The function deendianize performs the inverse.

Since we restrict to two's complement, and do not allow integer representations to contain padding bits, an $x \in \mathbb{Z}$ is an integer of type signed k in case $-2^{\mathsf{char_bits} * \mathsf{rank_size}\ k - 1} \leq x < 2^{\mathsf{char_bits} * \mathsf{rank_size}\ k - 1}$. An $x \in \mathbb{Z}$ is an integer of type unsigned k in case $0 \leq x < 2^{\mathsf{char_bits} * \mathsf{rank_size}\ k}$.

In order to deal with underspecification of operations, our interface not just contains a function int_binop to perform binary operations, but also a predicate $\mathsf{int_binop_ok}\, \tau\, op\, x\, y$ that describes when op is allowed to be performed on integers x and y of type τ. This is to allow both strict implementations that make integer overflow undefined, and those that let it wrap (as for example GCC with the `-fno-strict-overflow` flag and CompCert do). This predicate should be at least as strong as what is allowed by the C standard. Whenever an operation is allowed by the C standard, the result of $\mathsf{int_binop}\, \tau\, op\, x\, y$ should correspond to its specification by the standard.

Integer promotions/demotions should be handled explicitly using casts, for which we use a similar treatment as for operations.

Finally, a *C environment* consists of an integer implementation with integer ranks K, a valid typing environment Γ, and functions $\mathsf{sizeof} : \mathsf{type} \to \mathbb{N}_{>0}$ and $\mathsf{fieldsizes} : \mathsf{list\ type} \to \mathsf{list}\ \mathbb{N}$. These functions should satisfy:

$$
\begin{array}{lll}
\mathsf{sizeof}\ (si\ k) = \mathsf{rank_size}\ k & \mathsf{sizeof\ void} = 1 & \mathsf{sizeof}\ (\tau[n]) = n * \mathsf{sizeof}\ \tau \\
\mathsf{sizeof}\ (\mathsf{struct}\ s) = \Sigma\ \mathsf{fieldsizes}\ \vec{\tau} & \text{if } \Gamma\, s = \vec{\tau} & \\
\mathsf{sizeof}\ \tau_i \leq z_i & \text{for each } i < |\vec{\tau}| \text{ and } \mathsf{fieldsizes}\ \vec{\tau} = \vec{z} & \\
\mathsf{sizeof}\ \tau_i \leq \mathsf{sizeof}\ (\mathsf{union}\ u) & \text{for each } i < |\vec{\tau}| \text{ and } \Gamma\, u = \vec{\tau} &
\end{array}
$$

We define bitsizeof τ as sizeof $\tau \cdot \mathsf{char_bits}$. We let sizeof void $= 1$ so as to capture that a `void` pointer can point to individual bytes.

Although the definition of a C environment does not explicitly state anything about alignment, it is implicitly there. If an implementation has constraints on alignment, it should set up the function fieldsizes in such a way. Together with the dynamic typing constraints of the memory (as defined in Section 4) it is ensured that no improperly aligned stores and reads will occur.

Nita *et al.* describe a more concrete notion of a *C platform* than our notion of a C environment [13]. Important difference are that alignment is implicit in our definition, that we allow pointers $\tau*$ whose size can depend on τ, and that we restrict to 2's complement.

4 Bits, bytes and memory values

This section defines the internals of our memory model, and the representation of pointers. In the remainder of this paper we implicitly parametrize all definitions and proofs by a C environment with ranks K and typing environment Γ.

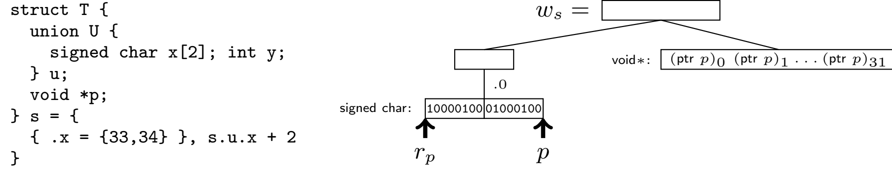

Fig. 2. A memory value w_s with pointer $p = (x_s, r_p, 2)_{\mathsf{signed\ short}>\mathsf{void}}$ on x86.

Definition 4.1. Bits*,* memory values*,* objects*, and* memories *are defined as:*

$$
\begin{aligned}
b \in \mathsf{bit} &::= \beta \mid (\mathsf{ptr}\ p)_i \mid \mathsf{indet} \\
w \in \mathsf{mval} &::= \mathsf{base}_{\tau_\mathsf{b}}\ \vec{b} \mid \mathsf{array}\ \vec{w} \mid \mathsf{struct}_s\ \vec{w} \mid \mathsf{union}_u\ (i, w) \mid \overline{\mathsf{union}}_u\ \vec{b} \\
o \in \mathsf{obj} &::= w \mid \mathsf{freed}\ \tau
\end{aligned}
$$

Memories ($m \in \mathsf{mem}$) *are finite partial functions of a countable set of* memory indexes ($x \in \mathsf{index}$) *to objects.*

A bit is either a concrete bit β (with β a Boolean), the ith fragment bit $(\mathsf{ptr}\ p)_i$ of a pointer p (see Definition 4.2 for pointers), or the indeterminate bit indet. As shown in Figure 2, integers are represented using concrete sequences of bits, and pointers as sequences of fragments. This way of representing pointers is similar to Leroy *et al.* [10], but is on the level of bits instead of bytes.

Memory values are decorated with types, so that we can read off the type $\mathsf{typeof}\ w$ of each memory value w. As empty arrays are prohibited, we do not store the element type of the $\mathsf{array}\ \vec{w}$ construct. We define the following partial function:

$$
\mathsf{indextype}_m\ x := \begin{cases} \mathsf{typeof}\ w & \text{if } m\,x = w \\ \tau & \text{if } m\,x = \mathsf{freed}\ \tau \end{cases}
$$

We consider two kinds of union values. The construct $\mathsf{union}_u\ (i, w)$ represents unions that are in a particular variant i, and the construct $\overline{\mathsf{union}}_u\ \vec{b}$ represents unions whose variant is unknown. Unions of the latter kind can be obtained by byte-wise copying, and will appear in uninitialized memory. Note that the variant of a union is internal to the memory model, and should not be exposed through the operational semantics (as an actual machine does not store it).

Leroy *et al.* [10] represent pointers as pairs (x, i) where x identifies the object in the whole memory, and i the offset into that object. Since we use trees as the contents of objects, we use paths through these trees to represent pointers.

Definition 4.2. References*,* addresses *and* pointers *are defined as:*

$$
\begin{aligned}
r \in \mathsf{ref} &::= \mathsf{T} \mid r \overset{s}{\rightsquigarrow} i \mid r \overset{u}{\rightsquigarrow}_q i \mid r \overset{n}{\rightsquigarrow} i \\
a \in \mathsf{addr} &::= (x, r, i)_{\tau > \sigma} \\
p \in \mathsf{ptr} &::= \mathsf{NULL}\ \tau \mid a
\end{aligned}
$$

References are paths from the top of a memory value to a subtree: the construct $r \overset{s}{\rightsquigarrow} i$ is used to take the ith field of a struct s, the construct $r \overset{s}{\rightsquigarrow}_q i$ to take the ith variant of a union u (the annotation $q \in \{\circ, \bullet\}$ will be explained on page 11), and the construct $r \overset{n}{\rightsquigarrow} i$ to take the i element of an array of length n. We use $r_1 +\!\!+ r_2$ to denote the concatenation of r_1 and r_2. We define $r : \tau \rightarrowtail \sigma$ to capture that r is a well-typed reference from type τ to σ.

In order to represent pointers, we have defined a richer structure than references, namely *addresses*. An address $(x, r, i)_{\tau>\sigma}$ consists of: (a) an object identifier x, (b) a reference r to a subtree of the memory value in the object at x, (c) an offset i to refer to a particular byte in the subtree at r (note that one cannot address individual bits in C), (d) the type τ of the subtree, and (e) the type σ to which the address is cast. The type τ is stored so we do not have to recompute it when performing a pointer cast.

Typing. We define typing judgments for all of the previously defined structures. As array indexing in C is performed using pointer arithmetic, we need some auxiliary operations on references to define the typing judgment of addresses.

$$\mathsf{refoffset}\ r := \begin{cases} i & \text{if } r = r' \overset{n}{\rightsquigarrow} i \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{refsize}\ r := \begin{cases} n & \text{if } r = r' \overset{n}{\rightsquigarrow} i \\ 0 & \text{otherwise} \end{cases}$$

$$r \oplus j := \begin{cases} r' \overset{n}{\rightsquigarrow} i + j & \text{if } r = r' \overset{n}{\rightsquigarrow} i \\ r & \text{otherwise} \end{cases}$$

Definition 4.3. *The typing judgment $m \vdash a : \sigma$ for addresses is defined as:*

$$\frac{\mathsf{refoffset}\ r = 0,\ \ r : \mathsf{indextype}_m\ x \rightarrowtail \tau,\ \ \tau > \sigma,\ \ i \leq \mathsf{sizeof}\ \tau \cdot \mathsf{refsize}\ r,\ \ \mathsf{sizeof}\ \sigma \mid i}{m \vdash (x, r, i)_{\tau>\sigma} : \sigma}$$

Here, $i \mid j$ means that i is a divisor of j. The relation $\tau > \sigma$, type τ is pointer castable to σ, *is the reflexive closure of: $\tau >$ unsigned char and $\tau >$ void.*

The premise $\mathsf{refoffset}\ r = 0$ ensures that r always points to the first element of an array subobject, the byte index i is then used to select an individual byte (if τ is unsigned char or void), or an element of the whole array. Adding j to $(x, r, i)_{\tau>\sigma}$ thus consists of changing the offset into $i + j \cdot \mathsf{sizeof}\ \sigma$ instead of moving r. Only when a pointer is dereferenced, or used for struct or union indexing, we use the normalized reference $r \oplus i \div \mathsf{sizeof}\ \sigma$.

An address remains well-typed after the object it points to has been deallocated (indextype is defined on freed objects as well). However, as addresses of deallocated objects are indeterminate [6, 6.2.4p2], we forbid them to be used for pointer arithmetic, *etc.* We use the non-strict inequality $i \leq \mathsf{sizeof}\ \tau \cdot \mathsf{refsize}\ r$ in the typing rule to allow addresses to be "one past the last element" [6, 6.5.6p8]. We call an address *strict* if is not "one past the last element" and its object has not been deallocated.

We define judgments $m \vdash p : \tau$ for pointers, $m \vdash b$ valid for bits, $m \vdash w : \tau$ for memory values, and $m \vdash o : \tau$ for objects. We display the rule for the construct

$\overline{\mathsf{union}}_u\,\vec{b}$ of a union whose variant is unknown for illustration.

$$\frac{\Gamma\,u = \vec{\tau} \qquad |\vec{\tau}| \neq 1 \qquad m \vdash \vec{b}\ \mathsf{valid} \qquad |\vec{b}| = \mathsf{bitsizeof}\ (\mathsf{union}\ u)}{m \vdash \overline{\mathsf{union}}_u\,\vec{b} : \mathsf{union}\ u}$$

We exclude unions with only one variant in this rule because their variant is always known. Validity of memories, notation m valid, is defined as:

$$\forall x\, w\,.\, m\,x = o \to \exists \tau\,.\, m \vdash o : \tau \ \wedge\ \mathsf{sizeof}\ \tau < 2^{\mathsf{char_bits}*(\mathsf{rank_size}\ \mathsf{ptr_rank})-1}$$

We need the restriction on the size to ensure that the result of pointer subtraction is representable by a signed integer of rank ptr_rank.

Conversion from and to bits. We compute the bit representation mtobits w of a memory value w by flattening it and inserting padding bits (as specified by fieldsizes). The bit representation of w_s displayed in Figure 2 is thus:

$$\mathsf{mtobits}\ w_s = 1000010001000100\ \mathsf{indet}\,\mathsf{indet}\ldots\mathsf{indet}\ (\mathsf{ptr}\ p)_0\ (\mathsf{ptr}\ p)_1\ldots(\mathsf{ptr}\ p)_{31}$$

Likewise, given a type τ and sequence of bits $\vec{b}$, we construct a memory value mofbits $\tau\ \vec{b}$ of type τ by iteration on τ (using Lemma 2.3). In the case of a union type u, we obviously cannot guess the variant as that information is not stored in the bit representation, so we use the $\overline{\mathsf{union}}_u\,\vec{b}$ construct.

Notice that mtobits and mofbits are neither left nor right cancellative. We do not have $\mathsf{mofbits}\ \tau\ (\mathsf{mtobits}\ w) = w$ for each $m \vdash w : \tau$ as variants of unions may have gotten lost, nor $\mathsf{mtobits}\ (\mathsf{mofbits}\ \tau\ \vec{b}) = \vec{b}$ for each $\vec{b}$ with $|\vec{b}| = \mathsf{bitsizeof}\ \tau$ as padding bits become indeterminate during the conversion.

Operations. In Section 6 we will define the following memory operations:

1. $\mathsf{alloc} : \mathsf{mem} \to \mathsf{index} \to \mathsf{type} \to \mathsf{mem}$ allocates a new object.
2. $\mathsf{free} : \mathsf{mem} \to \mathsf{index} \to \mathsf{mem}$ deallocates an object.
3. $_ \,!!\, _ : \mathsf{mem} \to \mathsf{addr} \to \mathsf{val}^{\mathsf{opt}}$ yields a stored value or fails in case it does not exist or effective types are violated.
4. $_[_ := _] : \mathsf{mem} \to \mathsf{addr} \to \mathsf{val} \to \mathsf{mem}$ stores a value.

Here, val is the data type of *abstract values* (see Section 5). Many of the above operations are partial, but are defined using a total function that assigns a default behavior to ease formalization. For example, alloc should only be used on fresh indexes, and $_[_ := _]$ should only be used if the address is accessible (*i.e.* $_ \,!!\, _$ succeeds). Notice that we model an unbounded memory as we consider a countable set of memory indexes. Formalizing a bounded memory is orthogonal to the strict-aliasing restrictions, and thus left for future work.

So as to define these operations, we first define variants on memory values, and lift those to whole memories in Section 6.

Definition 4.4. *The* empty memory value $\mathsf{new} : \mathsf{type} \to \mathsf{mval}$ *is defined as:*

$$\begin{aligned}
\mathsf{new}\ \tau_\mathsf{b} &:= \mathsf{base}_{\tau_\mathsf{b}}\,(\mathsf{indet} \ldots \mathsf{indet})\ (\mathsf{bitsizeof}\ \tau_\mathsf{b}\ \text{times})\\
\mathsf{new}\ (\tau[n]) &:= \mathsf{array}\,(\mathsf{new}\ \tau \ldots \mathsf{new}\ \tau)\ (n\ \text{times})\\
\mathsf{new}\ (\mathsf{struct}\ s) &:= \mathsf{struct}_s\,(\mathsf{new}\ \tau_0 \ldots \mathsf{new}\ \tau_{n-1}) \quad \text{if } \Gamma\, s = \tau_0 \ldots \tau_{n-1}\\
\mathsf{new}\ (\mathsf{union}\ u) &:= \begin{cases} \mathsf{union}_u\,(0, \mathsf{new}\ \tau) & \text{if } \Gamma\, u = \tau\\ \overline{\mathsf{union}}_u\,(\mathsf{indet} \ldots \mathsf{indet})\ (\mathsf{bitsizeof}\ (\mathsf{union}\ u)\ \text{times}) & \text{otherwise} \end{cases}
\end{aligned}$$

The operation new is used to create an empty memory value to implement alloc. The definition is well-defined for valid types by Lemma 2.3.

Definition 4.5. *The operation* $_ \mathbin{!!} _ : \mathsf{mval} \to \mathsf{ref} \to \mathsf{mval}^\mathsf{opt}$ *is defined as:*

$$\begin{aligned}
w \mathbin{!!} \top &:= w\\
(\mathsf{array}\,\vec{w}) \mathbin{!!} (\top \overset{n}{\rightsquigarrow} i +\!\!+ r) &:= w_i \mathbin{!!} r\\
(\mathsf{struct}_s\,\vec{w}) \mathbin{!!} (\top \overset{s}{\rightsquigarrow} i +\!\!+ r) &:= w_i \mathbin{!!} r\\
(\mathsf{union}_u\,(i, w)) \mathbin{!!} (\top \overset{u}{\rightsquigarrow}_q i +\!\!+ r) &:= w \mathbin{!!} r\\
(\mathsf{union}_u\,(j, w)) \mathbin{!!} (\top \overset{u}{\rightsquigarrow}_\bullet i +\!\!+ r) &:= (\mathsf{mofbits}\,\tau_i\,(\mathsf{mtobits}\ w)) \mathbin{!!} r \text{ if } \Gamma\, u = \vec{\tau},\ i \neq j\\
(\overline{\mathsf{union}}_u\,\vec{b}) \mathbin{!!} (\top \overset{u}{\rightsquigarrow}_q i +\!\!+ r) &:= (\mathsf{mofbits}\,\tau_i\,\vec{b}) \mathbin{!!} r \qquad \text{if } \Gamma\, u = \vec{\tau}
\end{aligned}$$

The look up operation is taking annotations q on union references $r \overset{s}{\rightsquigarrow}_q i$ into account: $q = \bullet$ means that we may access a union using a different variant than the current one (this is called *type-punning* [6, 6.5.2.3]), and $q = \circ$ means that this is prohibited. To enable type-punning, we convert back and forth to bits so as to interpret the memory value using a different type.

To ensure that type-punning is merely allowed when the memory is accessed "through the union type" [4], we change all annotations into $\circ$ whenever a pointer is stored in memory (see the definition of the function $\mathsf{btobits}$ in Section 5) or used as the argument of a function. This operation is called *freezing* a pointer, and the pointers whose annotations are all of the shape $\circ$ are called *frozen.* Frozen pointers cannot be used for type-punning by definition of $_ \mathbin{!!} _$.

The strict-aliasing restrictions [6, 6.5p6-7] state that an access affects the effective type of the accessed object. Since the word "access" covers both reads and stores [6, 3.1], this means that not only a store has a side-effects, but also a read. We factor these side-effects out using a function $\mathsf{force} : \mathsf{ref} \to \mathsf{mval} \to \mathsf{mval}$ that changes the effective types after a succeeded look up. To define the force and store operation, we define an auxiliary operation $\mathsf{alter}\ f : \mathsf{ref} \to \mathsf{mval} \to \mathsf{mval}$ that applies $f : \mathsf{mval} \to \mathsf{mval}$ to a subtree and changes the effective types accordingly. The interesting cases for unions are as follows (where $\Gamma\, u = \vec{\tau}$ and $i \neq j$):

$$\begin{aligned}
\mathsf{alter}\, f\,(\top \overset{u}{\rightsquigarrow}_q i +\!\!+ r)\,(\mathsf{union}_u\,(j, w)) &:= \mathsf{union}_u\,(i, \mathsf{alter}\, f\, r\,(\mathsf{mofbits}\ \tau_i\ (\mathsf{mtobits}\ w)))\\
\mathsf{alter}\, f\,(\top \overset{u}{\rightsquigarrow}_q i +\!\!+ r)\,(\overline{\mathsf{union}}_u\,\vec{b}) &:= \mathsf{union}_u\,(i, \mathsf{alter}\, f\, r\,(\mathsf{mofbits}\ \tau_i\ \vec{b}))
\end{aligned}$$

Now $\mathsf{force}\ r\ w := \mathsf{alter}\,(\lambda w' \,.\, w')\, r\, w$ and $w[r := w'] := \mathsf{alter}\,(\lambda _ \,.\, w')\, r\, w$.

5 Abstract values

The notion of memory values, as defined in the previous section, is quite low-level and exposes implementation-specific properties as bit representations. These details should remain internal to the memory model.

Definition 5.1. Base values *and* abstract values *are defined as:*

$$v_{\mathsf{b}} \in \mathsf{baseval} ::= \mathsf{indet}_{\tau_{\mathsf{b}}} \mid \mathsf{int}_{\tau_{\mathsf{i}}}\, i \mid \mathsf{ptr}\, p \mid \mathsf{byte}\, \vec{b}$$
$$v \in \mathsf{val} ::= v_{\mathsf{b}} \mid \mathsf{array}\, \vec{v} \mid \mathsf{struct}_s\, \vec{v} \mid \mathsf{union}_s\, (i, v) \mid \overline{\mathsf{union}}_u\, \vec{v}$$

Abstract values contain mathematical integers and pointers instead of bit arrays as their leafs. As fragment bits of pointers need to be kept outside of the memory when performing a byte-wise copy, the $\mathsf{byte}\, \vec{b}$ construct still exposes some low-level details. The typing rule for this construct is:

$$\frac{\text{Not all } \vec{b}\ \mathsf{indet} \qquad \text{Not all } \vec{b} \text{ of the shape } \beta \qquad m \vdash \vec{b}\ \mathsf{valid} \qquad |\vec{b}| = \mathsf{char_bits}}{m \vdash_{\mathsf{b}} \mathsf{byte}\, \vec{b} : \mathsf{unsigned\ char}}$$

This rule ensures that the $\mathsf{byte}\, \vec{b}$ is only used if $\vec{b}$ cannot be interpreted as an integer $\mathsf{int}_{\mathsf{unsigned\ char}}\, i$ or $\mathsf{indet}_{\mathsf{unsigned\ char}}$. The judgment $m \vdash_{\mathsf{b}} v_{\mathsf{b}} : \tau_{\mathsf{b}}$ moreover ensures that integers $\mathsf{int}_{\tau_{\mathsf{i}}}\, i$ are within range, and pointers $\mathsf{ptr}\, p$ are typed.

The function $\mathsf{base_binop} : \mathsf{binop} \to \mathsf{baseval} \to \mathsf{baseval} \to \mathsf{baseval}$ that performs a binary operation on base values is defined as:

$$\mathsf{base_binop}\ op\ (\mathsf{int}_{\tau_{\mathsf{b}}}\, i)\ (\mathsf{int}_{\tau_{\mathsf{b}}}\, j) := \mathsf{int}_{\tau}\ (\mathsf{int_binop}\ \tau_{\mathsf{b}}\ op\ i\ j)$$
$$\mathsf{base_binop}\ +\ (\mathsf{ptr}\, (x, r, i)_{\tau > \sigma})\ (\mathsf{int}_{\tau_{\mathsf{b}}}\, j) := \mathsf{ptr}\, (x, r, i + j \cdot \mathsf{sizeof}\ \sigma)_{\tau > \sigma}$$

and so on ..., together with a predicate $\mathsf{base_binop_ok} : \mathsf{binop} \to \mathsf{baseval} \to \mathsf{baseval} \to \mathsf{bool}$ that describes when it is allowed to perform the operation. Binary operations are prohibited on $\mathsf{indet}_{\tau_{\mathsf{b}}}$ and $\mathsf{byte}\, \vec{b}$ constructs.

Base values are converted into bit sequences as follows:

$$\mathsf{btobits}\ (\mathsf{indet}_{\tau_{\mathsf{b}}}) := \mathsf{indet} \ldots \mathsf{indet}\ (\mathsf{bitsizeof}\ \tau_{\mathsf{b}} \text{ times})$$
$$\mathsf{btobits}\ (\mathsf{int}_{\tau_{\mathsf{i}}}\, x) := \mathsf{endianize}\ (\tau_{\mathsf{i}}\text{-little endian representation of } x)$$
$$\mathsf{btobits}\ (\mathsf{ptr}\, p) := (\mathsf{ptr}\ (\mathsf{freeze}\, p))_0 \ldots (\mathsf{ptr}\ (\mathsf{freeze}\, p))_{\mathsf{bitsizeof}\ (\mathsf{typeof}\ p*) - 1}$$
$$\mathsf{btobits}\ (\mathsf{byte}\, \vec{b}) := \vec{b}$$

This function will be used to store values in memory (see Definition 6.1), hence we freeze pointers so as to avoid prohibited type-punning. The inverse function $\mathsf{bofbits}$ is defined in such a way that invalid bit patterns yield an $\mathsf{indet}_{\tau_{\mathsf{b}}}$.

Abstract values contain the construct $\overline{\mathsf{union}}_u\, \vec{v}$ for unions whose variant is unknown. The values $\vec{v}$ correspond to interpretations of all variants of u. Of course, these values should be consistent in the sense that they can be represented by the same bit sequence. The typing rule to ensure this is:

$$\frac{\Gamma\, u = \vec{\tau} \qquad |\vec{\tau}| \neq 1 \qquad m \vdash \vec{b}\ \mathsf{valid} \qquad |\vec{b}| = \mathsf{bitsizeof}\ (\mathsf{union}\ u) \qquad \forall i\, .\, v_i = \mathsf{vofbits}\ \tau_i\ \vec{b}}{m \vdash \overline{\mathsf{union}}_u\, \vec{v} : \mathsf{union}\ u}$$

The operation $\mathsf{vofbits}$ to obtain the bit representation of an abstract value, is defined similarly as its variant $\mathsf{mtobits}$ on memory values, but uses $\mathsf{bofbits}$ on the leafs. Obtaining a memory value $\mathsf{ofval}\ v$ and bit representation $\mathsf{vtobits}\ v$ from a value v is more challenging as the evidence of existence of a bit representation of a $\overline{\mathsf{union}}_u\ \vec{v}$ construct is only present in the typing judgment, and not in the value itself. We reconstruct the bits by "merging" the bit representations of all variants $\vec{v}$. For this, we define a join $\sqcup$ on bits satisfying $\mathsf{indet} \sqcup b = b$, $b \sqcup \mathsf{indet} = b$, and $b \sqcup b = b$. The case for the unknown union construct is

$$\mathsf{ofval}\ (\overline{\mathsf{union}}_u\ (v_0 \ldots v_{n-1})) := \overline{\mathsf{union}}_u\ (\mathsf{vtobits}\ v_0 \sqcup \ldots \sqcup \mathsf{vtobits}\ v_{n-1})$$

where $\sqcup$ is applied pointwise to the bit sequences obtained from $\mathsf{vtobits}$. This reconstruction is well-defined for well-typed abstract values.

6 The memory

Now that we have all definitions in place, we can finally combine them to define the main memory operations. In order to shorten these definitions we lift the operations $_ \ !! \ _$ and $_[_ := _]$ on memory values to whole memories. We define $m\ !!\ (x, r) := m\,x\ !!\ r$, and $m[(x, r) := w] := m[x := (m\,x)[r := w]]$. Notations are overloaded for conciseness of presentation.

Definition 6.1. *The* main memory operations *are defined as:*

$$m\ !!\ (x,r,i)_{\tau>\sigma} := \begin{cases} \textbf{let}\ \hat{r} := r \oplus i \div \mathsf{sizeof}\ \sigma,\ \ j := i \bmod \mathsf{sizeof}\ \sigma\ \textbf{in} \\ \textbf{if}\ \tau = \sigma\ \textbf{then}\ \mathsf{toval}\ (m\ !!\ (x,\hat{r})) \\ \textbf{else}\ \mathsf{vofbits}\ (\mathsf{unsigned\ char})\ (j\text{th byte of } m\ !!\ (x,\hat{r})) \end{cases}$$

$$m[(x,r,i)_{\tau>\sigma} := v] := \begin{cases} \textbf{let}\ \hat{r} := r \oplus i \div \mathsf{sizeof}\ \sigma,\ \ j := i \bmod \mathsf{sizeof}\ \sigma\ \textbf{in} \\ \textbf{if}\ \tau = \sigma\ \textbf{then}\ m[(x,\hat{r}) := \mathsf{ofval}\ v] \\ \textbf{else}\ m[(x,\hat{r}) := \text{set } j\text{th byte of } m\ !!\ (x,\hat{r}) \text{ to } \mathsf{vtobits}\ v] \end{cases}$$

$$\mathsf{force}\ (x,r,i)_{\tau>\sigma}\ m := \textbf{let}\ \hat{r} := r \oplus i \div \mathsf{sizeof}\ \sigma\ \textbf{in}\ m[x := \mathsf{force}\ \hat{r}\ (m\,x)]$$

$$\mathsf{alloc}\ x\ \tau\ m := m[x := \mathsf{new}\ \tau]$$

$$\mathsf{free}\ x\ m := m[x := \mathsf{freed}\ (\mathsf{indextype}_m\ x)]$$

The lookup operation $m\ !!\ (x, r, i)_{\tau>\sigma}$ normalizes the reference r, and then makes a case distinction on whether a whole subobject or a specific byte should be returned. In case of the former (*i.e.* $\tau = \sigma$), it converts the memory value $m\ !!\ (x, \hat{r})$ of the subobject in question into an abstract value. Otherwise, it yields an abstract value representing the jth byte of $m\ !!\ (x, \hat{r})$.

In the Coq development we have proved the expected laws about the interaction between the memory operations. We list some for illustration:

- If m valid, $m \vdash a : \tau$, and $m\ !!\ a = v$, then $m \vdash v : \tau$
- If m valid, $m \vdash a : \tau$, $m \vdash v : \tau$, and $m \vdash a' : \sigma$, then $m[a := v] \vdash a' : \sigma$

– If m valid, $a_1 \perp a_2$, $m \vdash a_2 : \tau_2$, $m \vdash v_2 : \tau_2$, and $m\ !!\ a_1 = v_1$, then $m[a_2 := v_2]\ !!\ a_1 = v_1$

Here, $a_1 \perp a_2$, denotes that a_1 and a_2 are *disjoint*, which 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.

Theorem 6.2 (Strict-aliasing). *Given a memory m with m valid, and frozen addresses $m \vdash a_1 : \sigma_1$ and $m \vdash a_2 : \sigma_2$ such that $\sigma_1, \sigma_2 \neq$ unsigned char and σ_1 not a subtype of σ_2 and vice versa. Now $a_1 \perp a_2$, or accessing a_1 after accessing a_2 and vice versa fails.*

Using this theorem, a compiler can optimize the generated code in the example below based on the assumption that p and q are not aliased.

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

If these pointers are aliased, the program exhibits undefined behavior as both the read from *q, and the assignment to *p, are considered an access (captured by the operations force and $_[_ := _]$ respectively).

In order to prove the correctness of program transformations one has to relate the memory states during execution of the original program to the memory states during execution of the transformed program. Leroy and Blazy [11] defined the notions of *memory extensions and injections* to facilitate this. We adapt memory extensions to our memory model, and demonstrate it by verifying an abstract version of the memcpy function that copies an object byte-wise.

A *memory extension* is a binary relation $\sqsubseteq$ on memories. The relation $m_1 \sqsubseteq m_2$ captures that m_2 makes more memory contents determinate, and that m_2 has fewer restrictions on effective types. This means that m_2 allows more behaviors. In order to define $\sqsubseteq$ we first define relations $\sqsubseteq_m$ on bits, abstract values, memory values, and objects. Some rules of these relations are:

$$\frac{}{b \sqsubseteq_m b} \qquad \frac{m \vdash b \ \mathsf{valid}}{\mathsf{indet} \sqsubseteq_m b} \qquad \frac{w_1 \sqsubseteq_m w_2}{\mathsf{union}_u\,(i, w_1) \sqsubseteq_m \mathsf{union}_u\,(i, w_2)} \qquad \frac{\vec{b}_1 \sqsubseteq_m \vec{b}_2}{\overline{\mathsf{union}}_u\,\vec{b}_1 \sqsubseteq_m \overline{\mathsf{union}}_u\,\vec{b}_2}$$

$$\frac{\Gamma\,u = \vec{\tau} \qquad |\vec{\tau}| \neq 1 \qquad w \sqsubseteq_m \mathsf{mofbits}\ \tau_i\ \vec{b} \qquad m \vdash \vec{b}\ \mathsf{valid} \qquad |\vec{b}| = \mathsf{bitsizeof}\ (\mathsf{union}\ u)}{\mathsf{union}_u\,(i, w) \sqsubseteq_m \overline{\mathsf{union}}_u\,\vec{b}}$$

The relation $m_1 \sqsubseteq m_2$ is now defined as for all x and o with $m_1\,x = o_1$ there exists an $o_2 \sqsupseteq_{m_2} o_1$ s.t. $m_2\,x = o_2$. This relation is a partial order. In order to use memory extensions to reason about program transformations we have to prove that all memory operations are respected by it. For example:

– If $w_1 \sqsubseteq_m w_2$ then mtobits $w_1 \sqsubseteq_m$ mtobits w_2
– If m_1 valid, $m_1 \sqsubseteq m_2$ and $m_1\ !!\ a = v_1$, then $\exists v_2 \sqsupseteq_{m_2} v_1$ s.t. $m_2\ !!\ a = v_2$

So as to show that a copy by assignment can be transformed into a byte-wise copy we proved that if $m \vdash w : \tau$, then ofval (toval w) $\sqsubseteq_m$ mofbits τ (mtobits w).

7 Formalization in Coq

Developing a formal version of a C11 memory model turned out to be much more difficult than we anticipated due to the complex and highly subtle nature of the aliasing restrictions introduced by the C99 and C11 standards. Hence, the use of a proof assistant has been essential for our development.

Since Coq is also a functional programming language, we can execute the memory model using it. This will be essential for the implementation of a certified interpreter in future work. We used Coq to formally prove properties such as:

- Type preservation and essential laws of the the memory operations.
- Compatibility of operations with respect to memory extensions.
- The fact that memory extensions form a partial order and respect typing.
- Correctness of an abstract `memcpy` and the Strict-aliasing Theorem 6.2.

We used Coq's notation mechanism combined with unicode symbols and type classes to let the Coq development correspond better to the definitions on paper. Type classes were also used to parametrize the whole development by an abstract interface for integer implementations and C environments (Section 3).

Although many operations on our memory model are partial, we formalized many such operations using a total function that assigns an appropriate default behavior. To account for partiality, we defined predicates that describe when these operations may be used. Alternatives include using the option monad or dependent types, but our approach turned out to be convenient as various proofs could be done easily by induction on the aforementioned predicate.

Our Coq code, available at `http://robbertkrebbers.nl/research/ch2o/`, is about 8.500 lines of code including comments and white space. Apart from that, we developed a library on general purpose theory (finite sets, finite functions, lists, the option monad, *etc.*) of about 10.000 lines.

8 Conclusion

The eventual goal of this work is to develop a formal semantics for a large part of the C11 programming language [8]. In previous work [9] we have developed a concise operational and axiomatic semantics for non-local control flow (goto and return statements). Recently, we have extended this work to include sequence points and non-deterministic expressions with side-effects [7]. The next step is to integrate our memory model into our operational semantics. Once integrated, we intend to develop a verified interpreter so we can test the memory model using actual C programs.

There are many other obvious extensions to our memory model: support for floating points, bit fields, variable length arrays, concurrency, *etc.* Bit fields are presumably easy to integrate as bits are already the smallest available unit in our memory model. Concurrency in C and C++ has received a lot of attention in formal developments (see *e.g.* Batty *et al.* [2]), but is extremely challenging

on its own. Treating the weaker aliasing restrictions on base types (*e.g.* reading a `signed int` using an `unsigned int`) is left for future work too.

In order to integrate the memory model into our axiomatic semantics based on separation logic [9], we have to be able to split memory objects into disjoint subobjects. This requires a disjoint union operation on memory values. Besides, the axiomatic semantics should take types seriously as our memory model is typed. The work of Tuch *et al.* [18] may be interesting for this even though they do not consider the aliasing restrictions of C.

Acknowledgments. I thank Freek Wiedijk, Herman Geuvers, Michael Nahas, and the anonymous referees for their helpful suggestions. I thank Xavier Leroy for many discussions on the CompCert memory model. This work is financed by the Netherlands Organisation for Scientific Research (NWO).

References

1. R. Affeldt and N. Marti. Towards formal verification of TLS network packet processing written in C. In *PLPV*, pages 35–46, 2013.
2. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.
3. C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.
4. GNU. GCC, the GNU Compiler Collection, 2011. `http://gcc.gnu.org/`.
5. International Organization for Standardization. WG14 Defect Report Summary, 2008. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/`.
6. International Organization for Standardization. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.
7. R. Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in C, 2013. Draft.
8. R. Krebbers and F. Wiedijk. A Formalization of the C99 Standard in HOL, Isabelle and Coq. In *CICM*, volume 6824 of *LNAI*, pages 297–299, 2011.
9. R. Krebbers and F. Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS*, volume 7794 of *LNCS*, pages 257–272, 2013.
10. X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, 2012.
11. X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR*, 41(1):1–31, 2008.
12. N. Maclaren. What is an Object in C Terms?, 2001. Mailing list message, `http://www.open-std.org/jtc1/sc22/wg14/9350`.
13. M. Nita, D. Grossman, and C. Chambers. A theory of platform-dependent low-level software. In *POPL*, pages 209–220, 2008.
14. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
15. T. Ramananandro, G. Dos Reis, and X. Leroy. Formal verification of object layout for C++ multiple inheritance. In *POPL*, pages 67–80, 2011.
16. V. Robert and X. Leroy. A Formally-Verified Alias Analysis. In *CPP*, volume 7679 of *LNCS*, pages 11–26, 2012.
17. J. G. Rossie and D. P. Friedman. An Algebraic Semantics of Subobjects. In *OOPSLA*, pages 187–199, 1995.
18. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL*, pages 97–108, 2007.