Separation algebras for C verification in Coq

Robbert Krebbers

ICIS, Radboud University Nijmegen, The Netherlands

Abstract. Separation algebras are a well-known abstraction to capture common structure of both permissions and memories in programming languages, and form the basis of models of separation logic. As part of the development of a formal version of an operational and axiomatic semantics of the C11 standard, we present a variant of separation algebras that is well suited for C verification.
Our variant of separation algebras has been fully formalized using the Coq proof assistant, together with a library of concrete implementations. These instances are used to build a complex permission model, and a memory model that captures the strict aliasing restrictions of C.

1 Introduction

Separation logic [19] is widely used to reason about imperative programs that use mutable data structures and pointers. Its key feature is the *separating conjunction* $P * Q$ that allows to split the memory into two disjoint parts; a part described by P, and another part described by Q. The separating conjunction is used for example in the *frame rule*.

$$\frac{\{P\}\, s\, \{Q\}}{\{P * R\}\, s\, \{Q * R\}}$$

This rule enables local reasoning about parts of a program. Given a Hoare triple $\{P\}\, s\, \{Q\}$, this rule makes it possible to derive that the triple also holds when the memory is extended with a disjoint part described by R.

In previous work, we have extended separation logic to deal with intricate features of the C programming language. In [15] we have extended separation logic to support non-local control flow in the presence of block scope variables (with pointers to those), and in [13] we have extended that separation logic to deal with non-determinism and sequence points in C.

A shortcoming of this first version of our separation logic for C is its rather basic memory model that merely supports integers and pointers, but no array, struct, and union types. In order to support these data-types together with the *strict-aliasing restrictions* of C11 [10, 6.5p6-7], which allow compilers to perform type-based alias analysis, one needs a rich memory model. For that reason, we have developed a memory model based on forests structured according to the shape of data types in C to accurately describe these restrictions [12].

In this paper, we will show how separation algebras are used for the integration of our separation logic and our memory model.

Separation logic for C. The key observation of our separation logic in [13] is the correspondence between non-determinism in expressions and a form of concurrency. Inspired by the rule for the parallel composition [18], we have the following kinds of rules for each operator $\circledcirc$.

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \qquad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q_1 * Q_2\}}$$

The intuitive idea of the above rule is that if the memory can be split up into two parts, in which the subexpressions e_1 respectively e_2 can be executed safely, then the full expression $e_1 \circledcirc e_2$ can be executed safely in the whole memory. Since the separating conjunction ensures that both parts of the memory do not have overlapping parts that will be written to, it is guaranteed that no interference of the side-effects of e_1 and e_2 occurs. It thus effectively rules out expressions as `(*p = 3) + (*p = 4)` that have undefined behavior [10, 6.5p2].

Our separation logic uses permissions [4], and therefore the singleton assertion has the shape $e_1 \overset{x}{\mapsto} e_2$ where x is the permission of the object e_2 at address e_1. Fractional permissions [5] are used to make sharing of read only memory of multiple subexpressions possible. This is needed in `*p + *p` for example.

Permissions are also used to keep track of whether an object has been locked due to a previous assignment. This is needed to ensure that no undefined behavior because of a sequent point violation occurs (modifying an object in memory more than once between two sequence points). Furthermore, since C only allows pointer arithmetic on addresses that exist (*i.e.* have not been deallocated), we need *existence permissions*. Bornat *et al.* [4] left existence permissions for future work, but our permission model incorporates these.

Since permissions are used to account for various constraints, they become very complex, especially when used in a memory model for a real-world language like C. We will use separation algebras to factor out common structure and to build the permission and memory model in a more compositional way.

Approach. Separation algebras, as originally defined by Calcagno *et al.* [6], are used as models of separation logic. Given a *separation algebra*, which is a partial cancellative commutative monoid $(A, \emptyset, \cup)$, a shallow embedding of separation logic with assertions $P, Q : A \to \mathsf{Prop}$ can be defined as:

$$\begin{aligned} \mathsf{emp} &:= \lambda x\,.\, x = \emptyset \\ P * Q &:= \lambda x\,.\, \exists x_1\, x_2\,.\, x = x_1 \cup x_2 \wedge P\, x_1 \wedge Q\, x_2 \end{aligned}$$

The prototypical instance of a separation algebra is a heap, where $\emptyset$ is the empty heap, and $\cup$ the disjoint union. Other useful instances include the booleans $(\mathsf{bool}, \mathsf{false}, \vee)$ and fractional permissions $([0,1]_{\mathbb{Q}}, 0, +)$ [4,5] where 0 denotes no access, 1 exclusive access, and $0 < \_ < 1$ read-only access. Separation algebras are closed under various many (products, finite functions, *etc.*), and hence complex instances can be built compositionally.

When formalizing separation algebras in the Coq proof assistant, we quickly ran into some problems:

1. Dealing with partial operations is cumbersome.
2. Dealing with subsets types (modeled as Σ-types) is inconvenient.
3. Operations like the difference operation $\setminus$ cannot be defined constructively from just the laws of a separation algebra.

To deal with problem 1 of partiality, we turn $\cup$ into a total binary operation, and axiomatize a binary relation $x \perp y$ that describes that x and y are *disjoint*. Only if $x \perp y$ holds, $x \cup y$ is required to satisfy the algebraic laws.

Problem 2 already appears in the simple case of fractional permissions $[0,1]_{\mathbb{Q}}$, where the $\cup$-operation (here $+$) can 'overflow'. We remedy this problem by having all operations operate on pre-terms (here $\mathbb{Q}$) and axiomatize a predicate valid that describes that a pre-term is valid (here $0 \leq \_ \leq 1$).

Although problems 1 and 2 seem relatively minor for trivial separation algebras like Booleans and fractional permissions, these problems become more evident for more complex (recursive) separation algebras like those that appear in our memory model. Our approach makes using plain ML/Haskell-style types possible. In order to deal with problem 3, we axiomatize the relation $\subseteq$ and the operation $\setminus$. Using a choice operator, the $\setminus$-operation can be defined in terms of $\cup$, but in Coq (without axioms) that is impossible.

Since the aforementioned problems merely concern ease of formalization, our solution so far is just a different form of presentation and does not fundamentally change the notion of a separation algebra. Although our solution results in more laws, these are generally trivial to prove. Moreover, we describe some machinery to deal with the additional conditions.

A more fundamental problem is that the standard definition of a separation algebra allows for very strange instances that do not correspond to a reasonable separation logic. To that end, Dockins *et al.* [8] have described various restrictions of separation algebras: splittability, positivity, disjointness, *etc.* Of course, we rather avoid the need to formalize a complex algebraic hierarchy in Coq. Hence, we define *one* variant that fits our whole development.

Our variant also includes additional features to abstractly describe exclusive ownership, which is needed for our permission and memory model.

Related work. Separation algebras were originally defined by Calcagno *et al.* [6], but their work dealt with a rather idealized language, and was not aimed at formalization in proof assistants. However, many researchers have used separation algebras and separation logic for realistic languages in proof assistants.

Dockins *et al.* [8] have formalized separation algebras together with various restrictions in Coq. They have dealt with the issue of partiality by treating $\cup$ as a relation instead of a function. However, this is unnatural, because equational reasoning becomes impossible and one has to name all auxiliary results.

Bengtson *et al.* [2] formalized separation algebras in Coq to reason about object-oriented programs. They have defined $\cup$ as a partial function, and did not define any complex permission models. Yet another formalization of separation algebras is by Klein *et al.* [11] using the Isabelle proof assistant. Their approach to partial operations is similar to ours. Section 2 contains a comparison.

In our previous paper on a separation logic for non-determinism and sequence points in C [13] we used an extension of *permission algebras* to describe permissions abstractly. Separation algebras are more general: the previous abstraction contained notions specific to permissions, and therefore the memory model itself was not an instance. Moreover, separation algebras include an $\emptyset$-element, which is necessary to split the trees of our memory model. The permission model that we present in this paper is based on our previous one [13], but it is built more compositionally, and it supports existence permissions.

There has been a significant amount of previous work on formalized memory models for the C programming language, most notably by Leroy *et al.* [17] and Beringer *et al.* [3] in the context of the CompCert compiler [16]. However, no previous memory models apart from our own [12] have taken the strict aliasing restrictions of C11 into account. Thus, in particular no previous work has dealt with separation logic for such a memory model.

Contribution. Our contribution is fivefold:

- We define a variant of separation algebras that works well in Coq (Section 2).
- We present a complex permission model for an operational and axiomatic semantics of the C programming language (Section 3 and 4).
- We present a generalization of the memory model that we described in [12] and show that it forms a separation algebra (Section 5).
- We present an algebraic method to reason about disjointness (Section 6).
- All proofs have been formalized using the Coq proof assistant (Section 7).

Because this paper is part of a large formalization effort, we often omit details and proofs. The interested reader can find all details online as part of the Coq development at `http://robbertkrebbers.nl/research/ch2o`.

2 Simple separation algebras

We first describe our version of separation algebras that is equivalent to traditional non-trivial, positive, and cancellative separation algebras.

Definition 2.1. *A* simple separation algebra *consists of a set A, with:*

- *An element* $\emptyset : A$
- *A predicate* $\mathsf{valid} : A \to \mathsf{Prop}$
- *Binary relations* $\bot,\ \subseteq\ : A \to A \to \mathsf{Prop}$
- *Binary operations* $\cup,\ \setminus\ : A \to A \to A$

Satisfying the following laws:

1. *If* $x \bot y$, *then* $y \bot x$ *and* $x \cup y = y \cup x$
2. *If* $\mathsf{valid}\ x$, *then* $\emptyset \bot x$ *and* $\emptyset \cup x = x$
3. *If* $x \bot y$ *and* $x \cup y \bot z$, *then*
   (a) $y \bot z$, $x \bot y \cup z$ *and*

(b) $x \cup (y \cup z) = (x \cup y) \cup z$
4. *If* $z \perp x$, $z \perp y$ *and* $z \cup x = z \cup y$, *then* $x = y$
5. *If* $x \perp y$, *then* $\mathsf{valid}\ x$ *and* $\mathsf{valid}\ (x \cup y)$
6. *There exists an* $x \neq \emptyset$ *with* $\mathsf{valid}\ x$
7. *If* $x \perp y$ *and* $x \cup y = \emptyset$, *then* $x = \emptyset$
8. *If* $x \perp y$, *then* $x \subseteq x \cup y$
9. *If* $x \subseteq y$, *then* $x \perp y \setminus x$ *and* $x \cup y \setminus x = y$

To deal with partiality, we turned $\cup$ into a total operation. Only if x and y are *disjoint*, notation $x \perp y$, we require $x \cup y$ to satisfy the algebraic laws.

Laws 2–4 describe the traditional laws of a separation algebra: identity, commutativity, associativity, and cancellativity. Law 5 ensures that valid is closed under $\cup$. Law 6 ensures that the separation algebra is non-trivial, together with law 5 this yields $\mathsf{valid}\ \emptyset$. Law 7 describes positivity, and laws 8 and 9 fully axiomatize the $\subseteq$-relation and $\setminus$-operation. Using positivity and cancellativity, we obtain that $\subseteq$ is a partial order and that $\cup$ is order preserving and respecting.

Definition 2.2. *The* simple Boolean separation algebra bool *is defined as:*

$$\begin{aligned} \mathsf{valid}\ x &:= \mathsf{True} & \emptyset &:= \mathsf{false} \\ x \perp y &:= \neg x \vee \neg y & x \cup y &:= x \vee y \\ x \subseteq y &:= x \to y & x \setminus y &:= x \wedge \neg y \end{aligned}$$

Boyland's fractional permissions $[0,1]_{\mathbb{Q}}$ [5] where 0 denotes no access, 1 exclusive access, and $0 < \_ < 1$ read-only access, form a simple separation algebra.

Definition 2.3. *The* simple fractional separation algebra $\mathbb{Q}$ *is defined as:*

$$\begin{aligned} \mathsf{valid}\ x &:= 0 \leq x \leq 1 & \emptyset &:= 0 \\ x \perp y &:= 0 \leq x, y \wedge x + y \leq 1 & x \cup y &:= x + y \\ x \subseteq y &:= 0 \leq x \leq y \leq 1 & x \setminus y &:= x - y \end{aligned}$$

The version of separation algebras by Klein *et al.* [11] in Isabelle also treats $\cup$ as a total operation and uses a relation $\perp$. There are some differences:

1. We include a predicate valid to prevent having to deal with subset types.
2. They have weaker premisses for associativity (law 3b), namely $x \perp y$, $y \perp z$ and $x \perp z$ instead of $x \perp y$ and $x \cup y \perp z$. Ours are more natural, *e.g.* for fractional permissions one has $0.5 \perp 0.5$ but not $0.5 + 0.5 \perp 0.5$, and it thus makes no sense to require $0.5 \cup (0.5 \cup 0.5) = (0.5 \cup 0.5) \cup 0.5$ to hold.
3. We axiomatize $\setminus$ because Coq does not have a choice operator.

3 Permissions and separation logic for C

Our semantics for the C programming language needs a complex permission system to account for whether certain operations are allowed or not. We classify the C permissions using the following *permission kinds*.

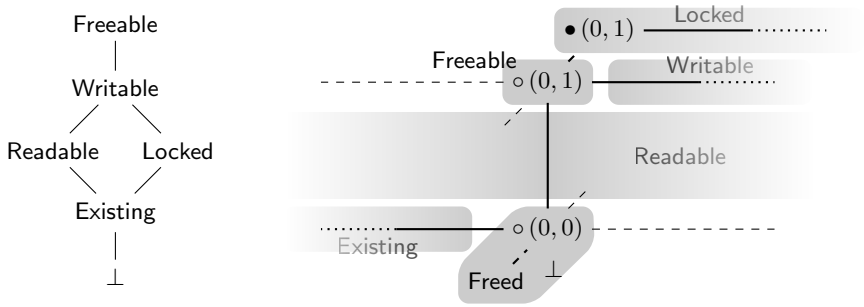

Fig. 1. Left: the lattice of permission kinds. Right: the actual permissions.

- Freeable. All operations (reading, writing, deallocation) are allowed.
- Writable. Just reading and writing is allowed.
- Readable. Solely reading is allowed.
- Existing. Objects with permissions of this kind are allowed to have pointers to them, which can be used for pointer arithmetic but cannot be dereferenced. Permissions of this kind are called *existence permissions* in [4].
- Locked. Permissions of this kind are used temporarily for objects that have been locked due to a write. The original permission of these objects will be restored at a subsequent sequence point [13].
  For example, in `(x = 3) + (*p = 4);` the assignment `x = 3` will lock the object `x`. The purpose of this lock is to describe the *sequence point restriction* of C that disallows to assign to the same object multiple times during the execution of the same expression. Hence, if `p` points to `x`, the expression will have undefined behavior. At the sequence point `;`, the object `x` will be unlocked, and its original permission will thereby be restored.
- $\bot$. No operations are allowed at all, and pointers to objects with permission of this kind are indeterminate. For example, `free(p); return (p-p);` has undefined behavior. After the call to `free`, the pointer `p` refers to an object with a permission of kind $\bot$. Therefore, `p` becomes indeterminate [10, 6.2.4p2], and cannot be used for pointer arithmetic anymore.

As displayed in Figure 1, permission kinds form a lattice $(\mathsf{pkind}, \subseteq)$ where $k_1 \subseteq k_2$ expresses that k_1 allows fewer operations than k_2. We use permission kinds as an abstract view of the permission model to allow the operational semantics to determine if certain operations are allowed. However, for our separation logic we have to deal with sharing. This is needed to:

- Split a Writable or Readable permission into Readable ones. This is needed in `x + x` where both parts require read ownership of `x`.
- Split a Freeable permission into an Existing and Writable one. This is needed in `*(p + 1) = *p = 1` where one part requires write ownership of `*p`, and another performs pointer arithmetic on `p` (which is only allowed if `*p` exists).

When reassembling split permissions (using $\cup$), we need to know when exclusive access is regained. Hence, the permission model needs to be more structured.

Definition 3.1. *A* C permissions system *is a separation algebra A with functions* $\mathsf{kind} : A \to \mathsf{pkind}$, $\mathsf{lock}, \mathsf{unlock}, \frac{1}{2} : A \to A$ *and* $\mathsf{token} : A$ *satisfying:*

$$\mathsf{unlock}\ (\mathsf{lock}\ x) = x \qquad \text{provided that } \mathsf{Writable} \subseteq \mathsf{kind}\ x \tag{1}$$

$$\mathsf{kind}\ (\mathsf{lock}\ x) = \mathsf{Locked} \qquad \text{provided that } \mathsf{Writable} \subseteq \mathsf{kind}\ x \tag{2}$$

$$\mathsf{kind}\ \left(\tfrac{1}{2}x\right) = \begin{cases} \mathsf{Readable} & \text{if } \mathsf{Writable} \subseteq \mathsf{kind}\ x \\ \mathsf{kind}\ x & \text{otherwise} \end{cases} \tag{3}$$

$$\mathsf{kind}\ \mathsf{token} = \mathsf{Existing} \tag{4}$$

$$\mathsf{kind}\ (x \setminus \mathsf{token}) = \begin{cases} \mathsf{Writable} & \text{if } \mathsf{kind}\ x = \mathsf{Freeable} \\ \mathsf{kind}\ x & \text{if } \mathsf{Existing} \subset \mathsf{kind}\ x \end{cases} \tag{5}$$

The $\frac{1}{2}$-operation is used to split a Writable or Readable permission x into two Readable permissions $\frac{1}{2}x$. Permissions of kind Locked cannot be split using $\frac{1}{2}$ because such permissions require exclusive write ownership. The $\setminus$-operation is used to take an *existence permission* token of some permission. In particular, it is used to split a Freeable permission x into an Existing permission token and Writable permission $x \setminus \mathsf{token}$. The existence permission token has kind Existing and thus allows solely pointer arithmetic.

A possible permission model satisfying these laws is (a subset of) the following three dimensional space:

$$\{\mathsf{Freed}\} + \{\circ, \bullet\} \times \mathbb{Q} \times [0,1]_{\mathbb{Q}}.$$

Figure 1 displays how the elements of this model project onto their kinds. This permission model combines fractional permissions to account for read/write ownership with counting permissions to account for the number of existence permissions (*i.e.* tokens) that have been handed out. The annotations $\{\circ, \bullet\}$ describe whether a permission is locked $\bullet$ or not $\circ$. Although counting permissions are traditionally modeled by natural numbers [4], our model uses rational numbers to allow the counting part to be splittable as well.

Our organization of permissions is inspired by CompCert [17], but has the additional Locked node. Since CompCert only deals with an operational semantics for C, it does not need to make a distinction between permissions and permission kinds. Therefore, a coarse permission model suffices.

4 Extended separation algebras

In this section we extend simple separation algebras with some features that will be used for our memory model. Moreover, we present various instances of separation algebras that will be used to contruct a C permission system.

Definition 4.1. *A* separation algebra *extends a simple separation algebra with:*

- *Predicates* $\mathsf{splittable}, \mathsf{unmapped}, \mathsf{unshared} : A \to \mathsf{Prop}$

– *A unary operation* $\frac{1}{2} : A \to A$

Satisfying the following laws:

10. If $x \perp x$*, then* $\mathsf{splittable}\ (x \cup x)$
11. If $\mathsf{splittable}\ x$*, then* $\frac{1}{2}x \perp \frac{1}{2}x$ *and* $\frac{1}{2}x \cup \frac{1}{2}x = x$
12. If $\mathsf{splittable}\ y$ *and* $x \subseteq y$*, then* $\mathsf{splittable}\ x$
13. If $x \perp y$ *and* $\mathsf{splittable}\ (x \cup y)$*, then* $\frac{1}{2}(x \cup y) = \frac{1}{2}x \cup \frac{1}{2}y$
14. $\mathsf{unmapped}\ \emptyset$*, and if* $\mathsf{unmapped}\ x$*, then* $\mathsf{valid}\ x$
15. If $\mathsf{unmapped}\ y$ *and* $x \subseteq y$*, then* $\mathsf{unmapped}\ x$
16. If $x \perp y$*,* $\mathsf{unmapped}\ x$ *and* $\mathsf{unmapped}\ y$*, then* $\mathsf{unmapped}\ (x \cup y)$
17. $\mathsf{unshared}\ x$ *iff* $\mathsf{valid}\ x$ *and for all* y *with* $x \perp y$ *we have* $\mathsf{unmapped}\ y$

The predicate unmapped describes whether storage with given permission is allowed to contain content or should be empty. Dually, unshared x describes whether a permission x has exclusive ownership of its storage. This means that all permissions disjoint to x do not allow their storage to contain content. The following table describes how the C permissions are classified using the predicates unmapped and unshared.

unshared	unmapped	Examples
		Readable permissions
	✓	The $\emptyset$ permission and Existing permissions
✓		Freeable, Writable and Locked permissions
✓	✓	The Freed permission

For separation algebras where unmapped and unshared make no sense (for example, the memory model in Section 5), we let $\mathsf{unmapped}\ x := x = \emptyset$ and $\mathsf{unshared}\ x := \mathsf{False}$. These definitions trivially satisfy laws 14–17.

The $\frac{1}{2}$-operation is partial because permissions without read ownership (for example those of kind Locked) cannot be split. Similar to the treatment of $\cup$, we turn $\frac{1}{2}$ into a total function and let splittable describe if a permission can be split (laws 10 and 11). Law 12 makes sure that splittable permissions are infinitely splittable, and law 13 ensures that $\frac{1}{2}$ distributes over $\cup$.

Definition 4.2. *The* Boolean separation algebra bool *is extended with:*

$$\mathsf{splittable}\ x := \neg x \qquad \tfrac{1}{2}x := x$$
$$\mathsf{unmapped}\ x := \neg x \qquad \mathsf{unshared}\ x := x$$

Definition 4.3. *The* fractional separation algebra $\mathbb{Q}$ *is extended with:*

$$\mathsf{splittable}\ x := 0 \leq x \leq 1 \qquad \tfrac{1}{2}x := 0.5 \cdot x$$
$$\mathsf{unmapped}\ x := x = 0 \qquad \mathsf{unshared}\ x := x = 1$$

A crucial part of the C permissions is the ability to lock permissions after an assignment to describe the sequence point restriction [13]. The *lockable separation algebra* adds annotations $\{\circ, \bullet\}$ to account for whether a permission is locked $\bullet$ or not $\circ$. Permissions that are locked have exclusive write ownership, and are thus only disjoint from those that do not allow content.

Definition 4.4. *Given a separation algebra A, the* lockable separation algebra $\mathcal{L}(A) := \{\circ, \bullet\} \times A$ over A *is defined as:*

$$\begin{aligned}
\mathsf{valid}\ (\circ\, x) &:= \mathsf{valid}\ x & \mathsf{valid}\ (\bullet\, x) &:= \mathsf{unshared}\ x \\
\emptyset &:= \circ\ \emptyset \\
\circ\, x \perp \circ\, y &:= x \perp y & \circ\, x \perp \bullet\, y &:= x \perp y \wedge \mathsf{unmapped}\ x \wedge \mathsf{unshared}\ y \\
\bullet\, x \perp \bullet\, y &:= \mathsf{False} & \bullet\, x \perp \circ\, y &:= x \perp y \wedge \mathsf{unshared}\ x \wedge \mathsf{unmapped}\ y \\
\circ\, x \cup \circ\, y &:= \circ\ (x \cup y) & \circ\, x \cup \bullet\, y &:= \bullet\ (x \cup y) \\
\bullet\, x \cup \bullet\, y &:= \bullet\ (x \cup y) & \bullet\, x \cup \circ\, y &:= \bullet\ (x \cup y)
\end{aligned}$$

We omitted the definition of some relations and operations in the previous and coming definitions due to space restrictions.

The C permission model needs existence permissions that allow pointer arithmetic but do not supply read or write ownership. The *counting separation algebra over A* has elements (x, y) with $x \in \mathbb{Q}$ and $y \in A$. Here, x counts the number of existence permissions that have been handed out. Existence permissions are elements $(x, \emptyset)$ with $x < 0$. To ensure that the counting separation algebra is closed under $\cup$ and preserves splittability, the counter x is rational.

Definition 4.5. *We let $z_{\mathbf{1}}$ and $z_{\mathbf{2}}$ denote the first and second projection of z.*

Definition 4.6. *Given a separation algebra A, the* counting separation algebra $\mathcal{C}(A) := \mathbb{Q} \times A$ over A *is defined as:*

$$\begin{aligned}
\mathsf{valid}\ x &:= \mathsf{valid}\ x_{\mathbf{2}} \wedge (\mathsf{unmapped}\ x_{\mathbf{2}} \to x_{\mathbf{1}} \leq 0) \wedge (\mathsf{unshared}\ x_{\mathbf{2}} \to 0 \leq x_{\mathbf{1}}) \\
\emptyset &:= (0, \emptyset) \\
x \perp y &:= x_{\mathbf{2}} \perp y_{\mathbf{2}} \wedge (\mathsf{unmapped}\ x_{\mathbf{2}} \to x_{\mathbf{1}} \leq 0) \wedge (\mathsf{unmapped}\ y_{\mathbf{2}} \to y_{\mathbf{1}} \leq 0) \\
&\qquad \wedge (\mathsf{unshared}\ (x_{\mathbf{2}} \cup y_{\mathbf{2}}) \to 0 \leq x_{\mathbf{1}} + y_{\mathbf{1}}) \\
x \cup y &:= (x_{\mathbf{1}} + y_{\mathbf{1}},\ x_{\mathbf{2}} \cup y_{\mathbf{2}})
\end{aligned}$$

Finally, we need to extend permissions with a permission Freed to keep track of whether storage has been deallocated. Deallocated storage is not allowed to contain any content, and pointers to deallocated storage are indeterminate and thereby cannot be used for pointer arithmetic.

Definition 4.7. *Given a separation algebra A, the* freeable separation algebra $\mathcal{F}(A) := \{\mathsf{Freed}\} + A$ over A *is defined by extending the separation algebra with:*

$$\begin{aligned}
\mathsf{valid}\ \mathsf{Freed} &:= \mathsf{True} & x \perp \mathsf{Freed} &:= x = \emptyset \\
\mathsf{Freed} \perp \mathsf{Freed} &:= \mathsf{False} & \mathsf{Freed} \perp y &:= y = \emptyset \\
& & x \cup \mathsf{Freed} &:= \mathsf{Freed} \\
\mathsf{Freed} \cup \mathsf{Freed} &:= \mathsf{Freed} & \mathsf{Freed} \cup y &:= \mathsf{Freed} \\
\mathsf{unmapped}\ \mathsf{Freed} &:= \mathsf{True} & \mathsf{unshared}\ \mathsf{Freed} &:= \mathsf{True}
\end{aligned}$$

Combining the previous separation algebras, we now define the C permission model. It is easy to verify that it satisfies the laws of Definition 3.1.

Definition 4.8. C permissions *are defined as*

$$\mathsf{perm} := \mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q})))$$

with:

$$\mathsf{kind}\ z := \begin{cases} \mathsf{Freeable} & \text{if } z = \circ\,(0,\,1) \\ \mathsf{Writable} & \text{if } z = \circ\,(x,\,1) \text{ with } x \neq 0 \\ \mathsf{Readable} & \text{if } z = \circ\,(x,\,y) \text{ with } 0 < y < 1 \\ \mathsf{Existing} & \text{if } z = \circ\,(x,\,0) \text{ with } x \neq 0 \\ \mathsf{Locked} & \text{if } z = \bullet\,(x,\,y) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{lock}\ z := \begin{cases} \bullet\,(x,\,y) & \text{if } z = \circ\,(x,\,y) \\ z & \text{otherwise} \end{cases}$$

$$\mathsf{unlock}\ z := \begin{cases} \circ\,(x,\,y) & \text{if } z = \bullet\,(x,\,y) \\ z & \text{otherwise} \end{cases}$$

$$\mathsf{token} := \circ\,(-1,\,0)$$

5 The C memory model and strict aliasing

In *type-based alias analysis*, type information is used to determine whether pointers are aliased or not. Consider:

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

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

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

A union is the C version of the sum type, but contrary to traditional sum types, unions are *untagged* instead of *tagged*. This means that the variant of a union cannot be obtained. Unions destroy the property that each memory area has a unique type that is statically known. The *effective type* [10, 6.5p6-7] of a memory area hence depends on the *run-time behavior* of the program.

The *strict-aliasing restrictions* of C11 [10, 6.5p6-7] ensure 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 has that particular variant. Calling `g` with aliased pointers (as in the example where `u` has the `y` variant, and is accessed through a pointer `p` to the `x` variant) results in undefined behavior.

Under certain circumstances it is nonetheless allowed to access a union using a pointer to another variant than its current one, this is called *type-punning* [10, 6.5.2.3]. For example, the function `g` has defined behavior (on architectures with `size_of(int)` $\leq$ `size_of(float)` and where `int`s do not have trap values):

```
int g() { union INT_FLT u; u.y = 3.0; return u.x; }
```

Type-punning may only be performed directly via an l-value of union type. The function `h` below thus exhibits undefined behavior because type-punning is performed indirectly via a pointer `p` to a variant of the union.

```
int h() { union INT_FLT u; int *p = &u.x; u.y = 3.0; return *p; }
```

Significant existing formal versions of C (*e.g.* those by Leroy *et al.* [17] and Ellison and Rosu [9]) model the memory as a finite partial function to objects, where each object consists of an array of bytes. Since these existing formal versions of C do not keep track of the variants of unions, they cannot capture the strict-aliasing restrictions of C11.

Instead of using an array of bytes to represent the contents of each object, our memory model [12] uses structured trees that have arrays of bits that represent base values (integers and pointers) on the leafs. This modification captures the strict-aliasing restrictions: effective types are modeled by the state of these trees.

A generalization of our memory model [12], where the leafs of the trees are elements of a separation algebra instead of just bits, forms a separation algebra. The original version of the memory model can be re-obtained by instantiating the generalized version with (permission annotated) bits.

Definition 5.1. C-trees *over a separation algebra A are defined as:*

$$w \in \mathsf{ctree}\, A ::= \mathsf{base}_{\tau_\mathsf{b}}\, \vec{x} \mid \mathsf{array}_\tau\, \vec{w} \mid \mathsf{struct}_s\, \overrightarrow{w\vec{x}} \mid \mathsf{union}_u\, (i, w, \vec{x}) \mid \overline{\mathsf{union}}_u\, \vec{x}$$

where $x \in A$. C-maps ($m \in \mathsf{cmap}\, A$) *are finite partial functions of a countable set of* memory indexes ($o \in \mathsf{index}$) *to pairs of booleans and C-trees.*

In the above definition, $s, u \in \mathsf{tag}$ range over struct and union names (called *tags*), $\tau_\mathsf{b} \in \mathsf{basetype}$ ranges over *base types* ($\mathsf{signed\ char}$, $\mathsf{unsigned\ int}$, $\tau*$, ...), and $\tau \in \mathsf{type}$ ranges over *types* (τ_b, $\tau[n]$, $\mathsf{struct}\ s$, $\mathsf{union}\ s$).

C-trees have two kinds of union nodes: $\mathsf{union}_u\, (i, w, \vec{x})$ represents a union in a particular variant i with padding $\vec{x}$, and $\overline{\mathsf{union}}_u\, \vec{x}$ represents a union whose variant is unknown. Unions of the latter kind can be obtained by byte-wise copying and appear in uninitialized memory. When accessing (reading or writing) a union $\overline{\mathsf{union}}_u\, \vec{x}$ using a pointer to variant i, the bits $\vec{x}$ will be interpreted as a C-tree w of variant i, and the node is changed into $\mathsf{union}_u\, (i, w, \vec{x'})$ where $\vec{x'}$ corresponds to the remaining padding. It is important to note that the variant of a union is internal to the memory model, and should not be exposed through the operational semantics because an actual machine does not store it.

Padding between struct fields is stored in the current version of our memory model, whereas it was absent in the original version [12]. For the actual instantiation, we have defined a predicate in the Coq formalization to ensure that padding always consists of indeterminate bits so as to be C11 compliant[1].

[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" [10, 6.2.6.1p6].

```
struct T {
  union U {
    signed char x[2]; int y;
  } u;
  void *p;
} s = {
  { .x = {33,34} }, s.u.x + 2
}
```

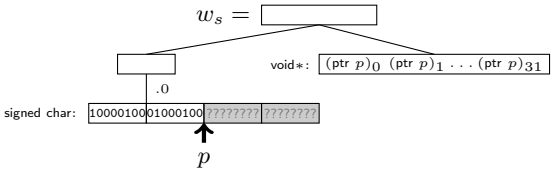

Fig. 2. The C-tree w_s corresponding to the object `s` declared in the C code on the left (on the x86 architecture). Permissions are omitted for simplicity.

The nodes (w, β) of C-maps are annotated with a boolean β to account for whether storage has been allocated dynamically using `malloc` (if $\beta = \mathsf{true}$) or statically as a block scope variable (if $\beta = \mathsf{false}$).

The original version of the memory model used specific nodes for objects that have been deallocated. In the current version we make it more uniform and represent such objects by a tree with Freed permissions at all leafs.

Definition 5.2. Bits *are defined as:*

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

where $p \in \mathsf{ptr}$ *ranges over pointers represented as paths through C-trees (see [12] for the formal definition).*

A bit is either a concrete bit 0 or 1, the ith fragment bit $(\mathsf{ptr}\ p)_i$ of a pointer p, or the indeterminate bit ?. As shown in Figure 2, integers are represented by concrete sequences of bits, and pointers by sequences of fragments. This way of representing pointers is similar to Leroy *et al.* [17], but is on the level of bits instead of bytes. The actual bit representation $\mathsf{flatten}\ w$ of a C-tree w is obtained by flattening it. For the C-tree w_s in Figure 2 we have:

$$\mathsf{flatten}\ w_s = 10000100\ 01000100\ ????????\ ????????\ (\mathsf{ptr}\ p)_0\ (\mathsf{ptr}\ p)_1 \ldots (\mathsf{ptr}\ p)_{31}$$

In order to re-obtain the actual memory model, we instantiate C-maps with permission annotated bits. For that, we use the *tagged separation algebra* that extends each element of an existing separation algebra with a tag.

Definition 5.3. *Given a separation algebra A and a set of tags T with default tag $t \in T$, the* tagged separation algebra $\mathcal{T}_{t:T}(A) := A \times T$ over A *is defined as:*

$$\begin{aligned}
\mathsf{valid}\ x &:= \mathsf{valid}\ x_{\mathbf{1}} \wedge (\mathsf{unmapped}\ x_{\mathbf{1}} \to x_{\mathbf{2}} = t)\\
\emptyset &:= (\emptyset, t)\\
x \perp y &:= x_{\mathbf{1}} \perp y_{\mathbf{1}} \wedge (\mathsf{unmapped}\ x_{\mathbf{1}} \vee x_{\mathbf{2}} = y_{\mathbf{2}} \vee \mathsf{unmapped}\ y_{\mathbf{1}})\\
&\qquad \wedge (\mathsf{unmapped}\ x_{\mathbf{1}} \to x_{\mathbf{2}} = t) \wedge (\mathsf{unmapped}\ y_{\mathbf{1}} \to y_{\mathbf{2}} = t)\\
x \cup y &:= \begin{cases} (x_{\mathbf{1}} \cup y_{\mathbf{1}},\ y_{\mathbf{2}}) & \text{if } x_{\mathbf{2}} = t\\ (x_{\mathbf{1}} \cup y_{\mathbf{1}},\ x_{\mathbf{2}}) & \text{otherwise} \end{cases}
\end{aligned}$$

The tagged separation algebra $\mathcal{T}_{t:T}(A)$ ensures that each element $x \in A$ with $\mathsf{unmapped}\ x$ element has the default tag t. For the case of permission annotated bits $\mathcal{T}_{?:\mathsf{bit}}(\mathsf{perm})$, we use the symbolic bit ? that represents indeterminate storage as the default tag to ensure that $\mathsf{unmapped}$ permissions have no content.

Definition 5.4. The C memory *is defined as:*

$$\mathsf{mem} := \mathsf{cmap}\,(\mathcal{T}_{?:\mathsf{bit}}(\mathcal{F}(\mathcal{L}(\mathcal{C}(\mathbb{Q}))))).$$

C-trees do not form a separation algebra because they do not have a single $\emptyset$ element (they have one for each type). However, apart from $\emptyset$ all other relations and operations can be defined by lifting those of the underlying separation algebra from the leafs to the trees. Defining the separation algebra structure on C-maps is then straightforward, the operations on the trees are lifted to finite functions, and the $\emptyset$ element is defined as the empty partial function.

The $\cup$-operation on (disjoint) C-trees is defined as follows:

$$\begin{aligned}
\mathsf{base}_{\tau_\mathsf{b}}\,\vec{x}_1 \cup \mathsf{base}_{\tau_\mathsf{b}}\,\vec{x}_2 &:= \mathsf{base}_{\tau_\mathsf{b}}\,(\vec{x}_1 \cup \vec{x}_2)\\
\mathsf{array}_\tau\,\vec{w}_1 \cup \mathsf{array}_\tau\,\vec{w}_2 &:= \mathsf{array}_\tau\,(\vec{w}_1 \cup \vec{w}_2)\\
\mathsf{struct}_s\,\overrightarrow{w_1\vec{x}_1} \cup \mathsf{struct}_s\,\overrightarrow{w_2\vec{x}_2} &:= \mathsf{struct}_s\,(\overrightarrow{w_1\vec{x}_1} \cup \overrightarrow{w_2\vec{x}_2})\\
\mathsf{union}_u\,(i, w_1, \vec{x}_1) \cup \mathsf{union}_u\,(i, w_2, \vec{x}_2) &:= \mathsf{union}_u\,(i, w_1 \cup w_2, \vec{x}_1 \cup \vec{x}_2)\\
\mathsf{union}_u\,(i, w_1, \vec{x}_1) \cup \overline{\mathsf{union}}_u\,\vec{x}_2 &:= \mathsf{merge}_\cup\,(\mathsf{union}_u\,(i, w_1, \vec{x}_1))\,\vec{x}_2\\
\overline{\mathsf{union}}_u\,\vec{x}_1 \cup \mathsf{union}_u\,(i, w_2, \vec{x}_2) &:= \mathsf{merge}_\cup\,(\mathsf{union}_u\,(i, w_2, \vec{x}_2))\,\vec{x}_1
\end{aligned}$$

Here, $\mathsf{merge}_f\ w\ \vec{x}$ yields a modified version of w in which the elements on the leaves are combined with $\vec{x}$ using the function f.

The above definition makes it possible to split storage of compound datatypes into smaller parts. However, splitting a union into a part with write ownership and a part with mere existence permissions is quite subtle because the variant of a union can change at run-time:

$$\begin{aligned}
\mathsf{union}_u\,(i, w_1, \vec{x_1}) &= \mathsf{union}_u\,(i, w_1', \vec{x_1'}) \cup \boxed{\text{part with existence permissions}}\\
&\Downarrow \quad \text{switching from variant } i \text{ to } j\\
\mathsf{union}_u\,(j, w_2, \vec{x}_2) &= \mathsf{union}_u\,(j, w_2', \vec{x_2'}) \cup \boxed{\text{part with existence permissions}}
\end{aligned}$$

Hence, for the part of a union that has mere existence permissions we always use the node $\overline{\mathsf{union}}_u\,\vec{x}$ with unknown variant. This restriction is enforced in the rules for disjointness and validity. Some representative rules are listed below:

$$\frac{\mathsf{valid}\ \vec{x}}{\mathsf{valid}\ (\overline{\mathsf{union}}_u\,\vec{x})} \qquad \frac{\mathsf{valid}\ w \quad \mathsf{valid}\ \vec{x} \quad \neg(\mathsf{unmapped}\ w \wedge \mathsf{unmapped}\ \vec{x})}{\mathsf{valid}\ (\mathsf{union}_u\,(i, w, \vec{x}))}$$

$$\frac{\mathsf{flatten}\ w_1 +\!\!+ \vec{x}_1 \perp \vec{x}_2 \quad \mathsf{valid}\ w_1 \quad \neg(\mathsf{unmapped}\ w_1 \wedge \mathsf{unmapped}\ \vec{x}_1) \quad \mathsf{unmapped}\ \vec{x}_2}{\mathsf{union}_u\,(i, w_1, \vec{x}_1) \perp \overline{\mathsf{union}}_u\,\vec{x}_2}$$

Since operations that change the variant (type-punning and byte-wise copying) are only allowed if the entire union has exclusive write ownership, the constraint $\mathsf{unmapped}\ \vec{x}_2$ ensures that disjointness is preserved under such operations.

6 Reasoning about disjointness

For the soundness proof of the axiomatic semantics in [13] we often had to reason about preservation of disjointness under memory operations. To ease that kind of reasoning, we have defined some machinery. In this section we will show that the machinery of [13] extends to any separation algebra.

Definition 6.1. Disjointness of a sequence $\vec{x}$, *notation* $\bot\,\vec{x}$, *is defined as:*

1. $\bot\,[\,]$
2. *If* $x \bot \bigcup \vec{x}$ *and* $\bot\,\vec{x}$, *then* $\bot\,(x :: \vec{x})$

Notice that $\bot\,\vec{x}$ is stronger than merely having $x_i \bot x_j$ for each $i \neq j$. For example, using fractional permissions, we do not have $\bot\,[0.5, 0.5, 0.5]$ whereas we clearly do have $0.5 \bot 0.5$. Using disjointness of sequences we can for example state the associativity law (law 3 of Definition 2.1) in a more symmetric way:

$$\text{if} \bot [x, y, z] \quad \text{then} \quad x \cup (y \cup z) = (x \cup y) \cup z.$$

Next, we define a relation $\vec{x}_1 \equiv_\bot \vec{x}_2$ that captures that sequences $\vec{x}_1$ and $\vec{x}_2$ behave equivalently with respect to disjointness.

Definition 6.2. Equivalence of $\vec{x}_1$ and $\vec{x}_2$ with respect to disjointness, *notation* $\vec{x}_1 \equiv_\bot \vec{x}_2$, *is defined as:*

$$\vec{x}_1 \leq_\bot \vec{x}_2 := \forall x .\bot\,(x :: \vec{x}_1) \to \bot\,(x :: \vec{x}_2)$$
$$\vec{x}_1 \equiv_\bot \vec{x}_2 := \vec{x}_1 \leq_\bot \vec{x}_2 \wedge \vec{x}_2 \leq_\bot \vec{x}_1$$

It is straightforward to show that $\leq_\bot$ is reflexive and transitive, is respected by concatenation of sequences, and is preserved by list containment. Hence, $\equiv_\bot$ is an equivalence relation, a congruence with respect to concatenation of sequences, and is preserved by permutations. The following results allow us to reason algebraically about disjointness.

Lemma 6.3. *If* $\vec{x}_1 \leq_\bot \vec{x}_2$, *then* $\bot\,\vec{x}_1$ *implies* $\bot\,\vec{x}_2$.

Lemma 6.4. *If* $\vec{x}_1 \equiv_\bot \vec{x}_2$, *then* $\bot\,\vec{x}_1$ *if and only if* $\bot\,\vec{x}_2$.

Theorem 6.5. *We have the following algebraic properties:*

$$\emptyset :: \vec{x} \equiv_\bot \vec{x} \tag{6}$$
$$(x_1 \cup x_2) :: \vec{x} \equiv_\bot x_1 :: x_2 :: \vec{x} \qquad \text{provided that } x_1 \bot x_2 \tag{7}$$
$$\bigcup \vec{x}_1 :: \vec{x}_2 \equiv_\bot \vec{x}_1 +\!\!+ \vec{x}_2 \qquad \text{provided that } \bot\,\vec{x}_1 \tag{8}$$
$$x_2 :: \vec{x} \equiv_\bot x_1 :: (x_2 \setminus x_1) :: \vec{x} \qquad \text{provided that } x_1 \subseteq x_2 \tag{9}$$

7 Formalization in Coq

All proofs in this paper have been fully formalized using Coq [7]. We used Coq's notation mechanism combined with unicode symbols and type classes to let the code correspond as well as possible to the definitions in this paper. The Coq development contains many parts that are not described in this paper, including the features of the original memory model [12].

In the Coq development, we used Coq's setoid machinery [20] to conveniently rewrite using the relations $\leq_\bot$ and $\equiv_\bot$ (see Definition 6.2). Using this, we have implemented a tactic that automatically solves entailments of the form:

$$H_0 :\bot \vec{x}_0,\ \ldots,\ H_n :\bot \vec{x}_n \quad \vdash \quad \bot \vec{x}$$

where $\vec{x}$ and $\vec{x}_i$ (for $i \leq n$) are arbitrary Coq expressions built from $\emptyset$, $\cup$ and $\bigcup$. This tactic works roughly as follows:

1. Simplify hypotheses using result 6-8 of Theorem 6.5.
2. Solve side-conditions by simplification using the same results and a solver for list containment (implemented by reflection).
3. Repeat these steps until no further simplification is possible.
4. Finally, solve the goal by simplification and list containment.

The Coq definitions corresponding to our memory model involve a lot of list surgery to translate between bit sequences and trees. To ease proofs about list surgery, we have developed a large library of general purpose theory. This library not only includes theory about lists, but also about finite sets, finite functions, and other data structures that are used heavily in the formalization.

8 Conclusions and further research

The eventual goal of this research is to develop an operational and axiomatic semantics (based on separation logic) for a large part of the C11 programming language [14]. This paper is an important step towards combining our separation logic [15,13] with our memory model [12]. However, a separation logic that can deal with the full (non-concurrent) C memory model remains future work.

For the operational semantics, one only needs a memory model that uses a coarse permission system, like the one used in CompCert [17]. In order to obtain a more concise operational semantics, one may therefore like to separate the memory model used in the operational semantics (with coarse permissions) from the one used in the axiomatic semantics (with rich permissions). The approach of *juicy memories* by Stewart and Appel [1, Chapter 42] might be useful.

It may be interesting to investigate what other permission models satisfy our requirements (see Definition 3.1). The permission model of Dockins *et al.* [8] may be a candidate.

Acknowledgments. I thank Freek Wiedijk and the anonymous referees for their helpful comments. This work is financed by NWO.

References

1. A. W. Appel, editor. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
2. J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq. In *ITP*, volume 6898 of *LNCS*, pages 22–38, 2011.
3. L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified Compilation for Shared-Memory C. In *ESOP*, volume 8410 of *LNCS*, pages 107–127, 2014.
4. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, 2005.
5. J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72, 2003.
6. C. Calcagno, P. W. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic. In *LICS*, pages 366–378, 2007.
7. Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2012.
8. R. Dockins, A. Hobor, and A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting. In *APLAS*, volume 5904 of *LNCS*, pages 161–177, 2009.
9. C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.
10. International Organization for Standardization. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.
11. G. Klein, R. Kolanski, and A. Boyton. Mechanised Separation Algebra. In *ITP*, volume 7406 of *LNCS*, pages 332–337, 2012.
12. R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*, 2013.
13. R. Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *POPL*, pages 101–112, 2014.
14. 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.
15. 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.
16. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
17. X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, 2012.
18. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, volume 3170 of *LNCS*, pages 49–67, 2004.
19. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001.
20. M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1), 2010.