An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C

Robbert Krebbers *
ICIS, Radboud University Nijmegen, The Netherlands
mail@robbertkrebbers.nl

Abstract

The C11 standard of the C programming language does not specify the execution order of expressions. Besides, to make more effective optimizations possible (*e.g.* delaying of side-effects and interleaving), it gives compilers in certain cases the freedom to use even more behaviors than just those of all execution orders.

Widely used C compilers actually exploit this freedom given by the C standard for optimizations, so it should be taken seriously in formal verification. This paper presents an operational and axiomatic semantics (based on separation logic) for non-determinism and sequence points in C. We prove soundness of our axiomatic semantics with respect to our operational semantics. This proof has been fully formalized using the Coq proof assistant.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Operational Semantics, Separation Logic, C Verification, Interactive Theorem Proving, Coq

1. Introduction

The C programming language [16, 17] is not only among the most popular programming languages in the world, but it is also among the most dangerous programming languages. Due to weak static typing and the absence of runtime checks, it is extremely easy for C programs to have bugs that make the program crash or behave badly in other ways. NULL-pointers can be dereferenced, arrays can be accessed outside their bounds, memory can be used after it is freed, *etc.* Furthermore, C programs can be developed with a too specific interpretation of the language in mind, giving portability and maintenance problems later.

Instead of forcing compilers to use a predefined execution order for expressions (*e.g.* left to right), the C standard does not specify it. This is a common cause of portability and maintenance problems, as a compiler may use an arbitrary execution order for each expression. Hence, to prove the correctness of a C program with respect to an arbitrary compiler, one has to verify that each possible execution order is legal and gives the correct result. To make more effective optimizations possible (*e.g.* delaying of side-effects and interleaving), the C standard requires the programmer to ensure that all execution orders satisfy certain conditions. If these conditions are not met, the program may do anything. Let us take a look at an example where one of those conditions is not met.

```
int main() {
  int x; int y = (x = 3) + (x = 4);
  printf("%d %d\n", x, y);
}
```

By considering all possible execution orders, one would naively expect this program to print `4 7` or `3 7`, depending on whether the assignment `x = 3` or `x = 4` is executed first. However, the *sequence point restriction* does not allow an object to be modified more than once (or being read after being modified) between two *sequence points* [16, 6.5p2]. A sequence point occurs for example at the end `;` of a full expression, before a function call, and after the first operand of the conditional `? :` operator [16, Annex C]. Hence, both execution orders lead to a sequence point violation, and are thus illegal. As a result, the execution of this program exhibits *undefined behavior*, meaning it may do literally anything.

The C standard uses a "garbage in, garbage out" principle for undefined behavior so that compilers do not have to generate (possibly expensive) runtime checks to handle corner cases. A compiler therefore does not have to generate code to test whether a sequence point violation occurs, but instead is allowed to assume no sequence point violations will occur and can use this information to perform more effective optimizations. Indeed, when compiled with `gcc -O1` (version 4.7.2), the above program prints `4 8`, which does not correspond to any of the execution orders.

Non-determinism in C is even more unrestrained than some may think. That the execution order in `e1 + e2` is unspecified, does not mean that either `e1` or `e2` will be executed entirely before the other. Instead, it means that execution can be interleaved; first a part of `e1`, then a part of `e2`, then another part of `e1`, and so on... Hence, the following expression is also allowed to print `bac`.

```
printf("a") + (printf("b") + printf("c"));
```

Many existing tools for C verification determinize the target program in one of their first phases, and let the user verify the correctness of the determinized version. When targeting a specific compiler for which the execution order is known, and which does not perform optimizations based on the sequence point restriction, this approach works. But to prove the correctness of a program with respect to an arbitrary compiler, this approach is insufficient even if all possible execution orders are considered (see the counterexample in `gcc` above).

* Part of this research has been done while the author was visiting INRIA-Paris Rocquencourt, France.

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01.
http://dx.doi.org/10.1145/2535838.2535878

Some verification tools perform syntactical checks to exclude sequence points violations. However, as it is undecidable whether a sequence point violation may occur [12], determinization combined with such checks is either unsound (slides 13 and 15 of Ellison and Rosu's POPL 2012 talk [13] present some examples in existing tools), or will exclude valid C programs.

Approach. As a step towards taking non-determinism and the sequence point restriction seriously in C verification, we extend the small step operational and separation logic for non-local control flow by Krebbers and Wiedijk [21] with non-deterministic expressions with side-effects and the sequence point restriction. Soundness of the axiomatic semantics is proved with respect to the operational semantics using the Coq proof assistant.

The straightforward approach to verification of programs with non-determinism is to consider all possible execution orders. However, naively this may result in a combinatorial explosion, and also it is not entirely clear how to use this approach in an axiomatic semantics that also handles the sequence point restriction. Hence, we take a different approach: we extend the axiomatic semantics with a Hoare judgment $\{P\}\, e\, \{Q\}$ for expressions. As usual, P is an assertion called the precondition. Like Von Oheimb [29], we let the postcondition Q be a function from values to assertions, because expressions not only have side-effects but primarily yield a value. Intuitively, the judgment $\{P\}\, e\, \{Q\}$ means that if P holds for the memory beforehand, and execution of e yields a value v, then $Q\, v$ holds for the resulting memory afterwards.

Besides partial program correctness, the judgment $\{P\}\, e\, \{Q\}$ ensures that e exhibits no undefined behavior due to the sequence point restriction. To deal with the unrestrained non-determinism in C, we observe that non-determinism in expressions corresponds to a form of concurrency, which separation logic is well capable of dealing with. Inspired by the rule for the parallel composition of separation logic (see [27]), we propose the following kind of rules for each operator $\circledcirc$.

$$\frac{\{P_l\}\, e_l\, \{Q_l\} \qquad \{P_r\}\, e_r\, \{Q_r\}}{\{P_l * Q_l\}\, e_l \circledcirc e_r\, \{P_r * Q_r\}}$$

The idea is that, if the memory can be split up into two disjoint parts (using the separating conjunction $*$), in which the subexpressions e_l respectively e_r can be executed safely, then the full expression $e_l \circledcirc e_r$ can be executed safely in the whole memory.

The actual rules of the axiomatic semantics (see Section 5) are more complicated. We have to deal with the return value, and have to account for undefined behavior due to integer overflow. To ensure no sequence point violations occur, we use separation logic with permission accounting. Like in ordinary separation logic with permissions [6], the singleton assertion becomes $e_1 \overset{\gamma}{\mapsto} e_2$ where γ is the permission of the object e_2 at address e_1, but we introduce a special class of *locked* permissions.

The inference rules of our axiomatic semantics are set up in such a way that reads and writes are only allowed for objects that are not locked, and moreover such that objects become locked after they have been written to. At constructs that contain a sequence point, the inference rules ensure that these locks are released. Fractional permissions [7] are used to allow memory that will not be written to be shared by multiple subexpressions.

Our approach to handling non-determinism and sequence points at the level of the operational semantics is inspired by Ellison and Rosu [12] and Norrish [25]. We annotate each object in memory with a permission, that is changed into a locked variant whenever a write occurs. This permission is changed back into the unlocked variant at the subsequent sequence point. Furthermore, we have a special state $\overline{\mathsf{undef}}$ for undefined behavior that is used for example whenever a write to a locked object occurs.

To bring the operational semantics closer to our axiomatic semantics, we make locks local to the subexpression where they were created. That means, at a sequence point we only unlock objects that have been locked by that particular subexpression, instead of unlocking all objects. This modification lets some artificial programs that were legal by the C standard exhibit undefined behavior (see page 5), but is not unsound for program verification.

An important part of both the operational and axiomatic semantics is the underlying permission system. We give an abstract specification of it (as an extension of permission algebras [8]), and give various instances of this abstraction.

Related work. Non-determinism, side-effects in expressions, and sequence points have been treated numerous times in formal treatments of C, but to our surprise, there is little evidence of work on program verification and program logics for these concepts.

The first formalization of a significant part of C is due to Norrish [25] using the proof assistant HOL4. An important part of his work was to accurately formalize non-determinism and sequence points as described by the C89 standard. He proved various Hoare rules for statements to facilitate reasoning about C programs. To reason about non-deterministic expressions, he proved that execution of sequence point free expressions is confluent [26]. This result is also useful for more efficient symbolic execution. Reasoning about arbitrary C expressions was left as an open problem.

Papaspyrou [30] has given a denotational semantics for a part of C89, including non-determinism and sequence points. He has implemented his semantics in Haskell to obtain a tool that can be used to explore all behaviors of a C program. Papaspyrou did not consider an axiomatic semantics.

More recently, Ellison and Rosu [12] have defined an executable semantics of the C11 standard using the $\mathbb{K}$-framework. Their semantics is very comprehensive and also describes non-determinism and the sequence point restriction. Moreover, since their semantics is effectively executable, it can be used as a debugger and an interpreter to explore all behaviors of a C program. It has been thoroughly tested against C test suites, and has been used by Regehr *et al.* to find bugs in widely used C compilers [31].

CompCert, a verified compiler C compiler by Leroy *et al.* written in Coq [23], supports non-determinism in expressions in the semantics of its source language. The interpreter by Campbell [9] can be used to explore this non-determinism. Krebbers [18] has extended CompCert's source language and interpreter with the sequence point restriction in the style of Ellison and Rosu [12].

There have been various efforts to verify programs in CompCert C. Appel and Blazy's axiomatic semantics for CompCert [2] operates on an intermediate language in which expressions have been determinized and side-effects have been removed. Their axiomatic semantics is thus limited to verification of programs compiled with CompCert, and will not work for arbitrary compilers. Herms [15] has formalized a verification condition generator based on the Why platform in Coq that can be used as a standalone tool via Coq's extraction mechanism. He proved the tool's soundness with respect to an intermediate language of CompCert, and it thus suffers from the same limitations as Appel and Blazy's work.

Black and Windley [5] have developed an axiomatic semantics for C. They define inference rules to factor out side-effects of expressions by translating these into semantically equivalent versions. Their axiomatic semantics seems rather limited, and soundness has not been proven with respect to an operational semantics.

Extending an axiomatic semantics with a judgment for expressions is not new, and has been done for example by Von Oheimb [29] for Java in the proof assistant Isabelle. His judgments for expressions are quite similar to ours, but his inference rules are not. Since he considered Java, he was able to use that the execution order of expressions is fully defined, which is not the case for C.

Contribution. Our contribution is fourfold:

- We define an abstract interface for permissions on top of which the memory model is defined, and present an algebraic method to reason about disjoint memories (Section 2).
- We define a small step operational semantics that handles non-determinism and sequence points (Section 3 and 4).
- We give an axiomatic semantics that allows reasoning about programs with non-determinism. This axiomatic semantics ensures that no undefined behavior (*e.g.* sequence point violations and integer overflow) occurs (Section 5).
- We prove the soundness of our axiomatic semantics (Section 6). This proof, together with some extensions (Section 7), has been fully formalized using Coq (Section 8).

As this paper describes a large formalization effort, we often omit details and proofs. The interested reader can find all details online as part of our Coq formalization.

2. The memory and permissions

We model memories as finite partial functions from some countable set of *memory indices* ($b \in \mathsf{index}$) to pairs of values and permissions. A value is either indet (which is used for uninitialized memory and the return value of functions without explicit return), a bounded integer, a pointer, or a NULL-pointer. Values are untyped (apart from integer values) and intentionally kept simple to focus on the issues of the paper.

DEFINITION 2.1. *A* partial function *from A to B is a (total) function from A to B^{opt}, where B^{opt} is the* option type*, defined as containing either $\bot$ or x for some $x \in B$. A partial function is called* finite *if its domain is finite. The operation $f[x := y]$ stores the value y at index x, and $f[x := \bot]$ deletes the value at index x.*

DEFINITION 2.2. Integer types *and* values *are defined as:*

$$\tau \in \mathsf{inttype} ::= \mathsf{signed\ char} \mid \mathsf{unsigned\ char} \mid \mathsf{signed\ int} \mid \ \ldots$$
$$v \in \mathsf{val} ::= \mathsf{indet} \mid \mathsf{int}_\tau\, n \mid \mathsf{ptr}\, b \mid \mathsf{NULL}$$

For an integer value $\mathsf{int}_\tau\ n$, the mathematical integer $n \in \mathbb{Z}$ should be within the bounds of τ. A value v is true*, notation $\mathsf{istrue}\ v$, if it is of the shape $\mathsf{int}_\tau\ n$ with $n \neq 0$, or $\mathsf{ptr}\, b$. It is* false*, notation $\mathsf{isfalse}\ v$, if it is of the shape $\mathsf{int}_\tau\ 0$ or NULL.*

Notice that indet is neither true nor false, because at an actual machine uninitialized memory has arbitrary contents.

In order to abstract from a concrete choice for a permission system (no permissions, simple read/write/free permissions, fractional permissions, *etc.*) in the definition of the memory, we define an abstract interface for permissions. We organize permissions using four different *permission kinds* (pkind):

- Free, which allows all operations (read, write, free),
- Write, which allows just reading and writing,
- Read, which allows solely reading, and
- Locked, which is temporarily used to lock an object between a write to it and a subsequent sequence point.

This organization is inspired by Leroy *et al.* [24], but differs by the fact that we abstract away from a concrete implementation and allow much more complex permission systems (*e.g.* those based on fractional permissions). These are needed for our separation logic.

We define a partial order on permission kinds as the reflexive-transitive closure of $\mathsf{Read} \subseteq \mathsf{Write}$, $\mathsf{Locked} \subseteq \mathsf{Write}$ and $\mathsf{Write} \subseteq \mathsf{Free}$. Since read-only memory cannot be used for writing, and therefore cannot be locked, the kinds Locked and Read are incomparable on purpose.

DEFINITION 2.3. *A* permission system *consists of a set P, binary relations $\subseteq$ and $\bot$, binary operations $\cup$ and $\setminus$, and functions $\mathsf{kind} : P \to \mathsf{pkind}$ and $\mathsf{lock}, \mathsf{unlock} : P \to P$, satisfying:*

1. *$(P, \subseteq)$ is a partial order*
2. *$\bot$ is symmetric*
3. *$(P, \cup)$ is a commutative semigroup*
4. *If $x \subseteq y$, then $\mathsf{kind}\ x \subseteq \mathsf{kind}\ y$*
5. *If $x \bot y$, then $\mathsf{kind}\ x = \mathsf{Read}$*
6. *If $\mathsf{Write} \subseteq \mathsf{kind}\ x$, then $\mathsf{unlock}\ (\mathsf{lock}\ x) = x$*
7. *If $\mathsf{kind}\ x \neq \mathsf{Locked}$, then $\mathsf{unlock}\ x = x$*
8. *$\mathsf{kind}\ (\mathsf{unlock}\ x) \neq \mathsf{Locked}$*
9. *If $x \bot y$ and $x' \subseteq x$, then $x' \bot y$*
10. *If $x \cup y \bot z$, then $x \bot z$ and $x \bot y \cup z$*
11. *If $x \bot y$, then $x \subset x \cup y$*
12. *If $x \subseteq y$, then $z \cup x \subseteq z \cup y$*
13. *If $z \bot x$, $z \bot y$, and $z \cup x \subseteq z \cup y$, then $x \subseteq y$*
14. *If $x \subset y$, then $x \bot y \setminus x$ and $x \cup y \setminus x = y$*

Permission systems extend *permission algebras* by Calcagno *et al.* [8] by organizing permissions using kinds, and by having operations for locking and unlocking. Whereas $\cup$ is a partial function in a permission algebra, we require it to be a total function, and account for partiality using a relation $\bot$ to describe that two permissions are disjoint and may be joined. For permissions that are not disjoint, the result of $\cup$ is not specified in the definition of a permission system. This relieves us from dealing with partial functions in Coq. Lastly, we require $\setminus$ to be a primitive so we can lift it to an operation on memories (see Definition 2.9) that is an actual Coq function.

Dockins *et al.* [11] remedy the issue of partially by defining $\cup$ as a relation instead of a function. But as for the operation $\setminus$, we prefer to use functions to ease reasoning about memories.

Rule 5 ensures that only permissions whose kind is Read are disjoint. This is to ensure that only readable parts of disjoint memories may overlap.

Before we define memories, we define three instances of permission systems. We begin with fractional permissions [7].

DEFINITION 2.4. Fractional permissions ($z \in \mathsf{frac}$) *are rational numbers within $(0, 1]$. We let $z_1 \subseteq z_2$ iff $z_1 \leq z_2$, and $z_1 \bot z_2$ iff $z_1 + z_2 \leq 1$. The operations are defined as:*

$$z_1 \cup z_2 := \begin{cases} z_1 + z_2 & \text{if } z_1 + z_2 \leq 1 \\ 1 & \text{otherwise} \end{cases} \qquad \mathsf{kind}\ z := \begin{cases} \mathsf{Write} & \text{if } z = 1 \\ \mathsf{Read} & \text{otherwise} \end{cases}$$
$$z_1 \setminus z_2 := \begin{cases} z_1 - z_2 & \text{if } 0 < z_1 - z_2 \\ 1 & \text{otherwise} \end{cases} \qquad \mathsf{lock}\ z := \mathsf{unlock}\ z := z$$

Notice that in the above definition we yield the dummy value 1 for $z_1 \cup z_2$ if $z_1 \not\perp z_2$, and $z_1 \setminus z_2$ if $z_2 \not\subseteq z_1$.

To account for locks due to the sequence point restriction, we extend fractional permissions with a special permission Seq.

DEFINITION 2.5. Sequenceable permissions *are defined as:*

$$\gamma_{\mathsf{s}} \in \mathsf{seqfrac} ::= \mathsf{Seq} \mid \mathsf{UnSeq}\ z$$

Disjointness $\bot$ is inductively defined as:

1. *if $z_1 \bot z_2$ then $\mathsf{UnSeq}\ z_1 \bot \mathsf{UnSeq}\ z_2$*

The order $\subseteq$ is inductively defined as:

1. *$\mathsf{Seq} \subseteq \mathsf{Seq}$*
2. *if $z_1 \subseteq z_2$, then $\mathsf{UnSeq}\ z_1 \subseteq \mathsf{UnSeq}\ z_2$*

The operations $\cup$ and $\setminus$ are defined point-wise, and:

$$\mathsf{kind}\ \mathsf{Seq} := \mathsf{Locked} \qquad \mathsf{kind}\ (\mathsf{UnSeq}\ z) := \mathsf{kind}\ z$$
$$\mathsf{lock}\ \mathsf{Seq} := \mathsf{Seq} \qquad \mathsf{lock}\ (\mathsf{UnSeq}\ z) := \mathsf{Seq}$$
$$\mathsf{unlock}\ \mathsf{Seq} := \mathsf{UnSeq}\ 1 \qquad \mathsf{unlock}\ (\mathsf{UnSeq}\ z) := \mathsf{UnSeq}\ z$$

We extend sequenceable permissions to account for some subtleties of C. First of all, we need to deal with objects that are declared using the `const` qualifier (those are read-only). Secondly, whereas dynamically allocated memory obtained via `alloc` can be freed manually using `free`, memory of block scope variables cannot be freed using `free`. Freeing it should therefore be prohibited by the permission system.

DEFINITION 2.6. Full permissions *are defined as:*

$$\gamma \in \mathsf{perm} ::= \mathsf{Freeable}\ \gamma_\mathsf{s} \mid \mathsf{Writable}\ \gamma_\mathsf{s} \mid \mathsf{ReadOnly}\ z$$

Here, all relations and operations are defined point-wise by lifting those on fractional and sequenceable permissions. We use the abbreviations F, W *and* R *for the permissions* $\mathsf{Freeable}\ (\mathsf{UnSeq}\ 1)$, $\mathsf{Writable}\ (\mathsf{UnSeq}\ 1)$ *and* $\mathsf{ReadOnly}\ 1$*, respectively.*

Given an arbitrary permission system with carrier P, the definition of the memory is now straightforward.

DEFINITION 2.7. Memories *($m \in \mathsf{mem}$) are finite partial functions from memory indices to pairs* (v, x) *with* $v \in \mathsf{val}$ *and* $x \in P$.

First we define the operations that are used by the operational semantics (Section 4) to interact with the memory.

DEFINITION 2.8. *The* memory operations *are defined as:*

$$\mathsf{perm}\ b\ m := \begin{cases} x & \text{if } m\,b = (v,\ x) \\ \bot & \text{otherwise} \end{cases}$$

$$m\ !!\ b := \begin{cases} v & \text{if } m\,b = (v,\ x) \text{ and } \mathsf{kind}\ x \neq \mathsf{Locked} \\ \bot & \text{otherwise} \end{cases}$$

$$m[b := v] := \begin{cases} m[b := (v,\ x)] & \text{if } m\,b = (v',\ x) \\ m & \text{otherwise} \end{cases}$$

$$\mathsf{alloc}\ b\ v\ x\ m := m[b := (v,\ x)]$$

$$\mathsf{free}\ b\ m := m[b := \bot]$$

$$\mathsf{locks}\ m := \{b \mid m\,b = (v,\ x) \wedge \mathsf{kind}\ x = \mathsf{Locked}\}$$

$$\mathsf{lock}\ b\ m := \begin{cases} m[b := (v,\ \mathsf{lock}\ x)] & \text{if } m\,b = (v,\ x) \\ m & \text{otherwise} \end{cases}$$

$$\mathsf{unlock}\ \Omega\ m := \{(b, (v, \mathsf{unlock}\ x)) \mid b \in \Omega \wedge m\,b = (v,\ x)\} \cup \{(b, (v, x)) \mid b \notin \Omega \wedge m\,b = (v,\ x)\}$$

Here, we have $\Omega \subseteq_{\mathsf{fin}} \mathsf{index}$.

The function perm is used to obtain the permission of an object. Allocation $\mathsf{alloc}\ b\ v\ x\ m$ of an object with value v and permission x should only be used with unused indices b in m (*i.e.* with $\mathsf{perm}\ b\ m = \bot$). Likewise, a store $\_[\_ := \_]$ and deallocation using free, should only be used when the permissions are appropriate. We will take care of these side-conditionals in the rules of the operational semantics. Next, we define the memory operations that are used by the axiomatic semantics (Section 5).

DEFINITION 2.9. *The* separation memory relations *are defined as:*

- *We let* $m_1 \perp m_2$ *iff for all* b, v_1, x_1, v_2 *and* x_2 *with* $m_1\,b = (v_1,\ x_2)$ *and* $m_2\,b = (v_2,\ x_2)$ *we have* $v_1 = v_2$ *and* $x_1 \perp x_2$.
- *We let* $m_1 \subseteq m_2$ *iff for all* b, v_1 *and* x_1 *with* $m_1\,b = (v,\ x_1)$ *there exists an* $x_2 \subseteq x_1$ *with* $m_2\,b = (v,\ x_2)$.

The separation memory operations *are defined as:*

$$m_1 \cup m_2 := \{(b, (v_1, x_1 \cup x_2)) \mid m_1\,b = (v_1,\ x_1) \wedge m_2\,b = (v_2,\ x_2)\} \cup \{(b, (v_1, x_1)) \mid m_1\,b = (v_1,\ x_1) \wedge m_2\,b = \bot\} \cup \{(b, (v_2, x_2)) \mid m_1\,b = \bot \wedge m_2\,b = (v_2,\ x_2)\}$$

$$m_1 \setminus m_2 := \{(b, (v_1, x_1 \setminus x_2)) \mid m_1\,b = (v_1,\ x_1) \wedge m_2\,b = (v_2,\ x_2) \wedge x_2 \subset x_1\} \cup \{(b, (v_1, x_1)) \mid m_1\,b = (v_1,\ x_1) \wedge m_2\,b = \bot\}$$

The union $\cup$ and empty memory $\emptyset$ form a monoid that is commutative and cancellative for disjoint memories.

For the soundness proof of our axiomatic semantics (Section 6) we often need to reason about preservation of disjointness under memory operations. To ease that kind of reasoning, we define a relation $\vec{m}_1 \equiv_\perp \vec{m}_2$ that describes that the memories $\vec{m}_1$ and $\vec{m}_2$ behave equivalently with respect to disjointness.

DEFINITION 2.10. Disjointness of a list of memories $\vec{m}$*, notation* $\perp \vec{m}$*, is inductively defined as:*

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

Notice that $\perp \vec{m}$ is stronger than merely having $m_i \perp m_j$ for each $i \neq j$. For example, using fractional permissions, we do not have $\perp [\,\{(b, (v, 0.5))\}, \{(b, (v, 0.5))\}, \{(b, (v, 0.5))\}\,]$, whereas we clearly do have $\{(b, (v, 0.5))\} \perp \{(b, (v, 0.5))\}$.

DEFINITION 2.11. Equivalence of $\vec{m}_1$ and $\vec{m}_2$ with respect to disjointness*, notation* $\vec{m}_1 \equiv_\perp \vec{m}_2$*, is defined as:*

$$\vec{m}_1 \leq_\perp \vec{m}_2 := \forall m\,.\, \perp (m :: \vec{m}_1) \to \perp (m :: \vec{m}_2)$$

$$\vec{m}_1 \equiv_\perp \vec{m}_2 := \vec{m}_1 \leq_\perp \vec{m}_2 \wedge \vec{m}_2 \leq_\perp \vec{m}_1$$

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

FACT 2.12. *If* $\vec{m}_1 \leq_\perp \vec{m}_2$ *and* $\perp \vec{m}_1$*, then* $\perp \vec{m}_2$.

THEOREM 2.13. *We have the following algebraic properties:*

1. $\emptyset :: \vec{m} \equiv_\perp \vec{m}$
2. $(m_1 \cup m_2) :: \vec{m} \equiv_\perp m_1 :: m_2 :: \vec{m}$ *provided that* $m_1 \perp m_2$
3. $\bigcup \vec{m}_1 :: \vec{m}_2 \equiv_\perp \vec{m}_1 +\!\!+ \vec{m}_2$ *provided that* $\perp \vec{m}_1$
4. $m[b := v] :: \vec{m} \equiv_\perp m :: \vec{m}$ *provided that* $\mathsf{perm}\ b\ m = x$ *for some* x *that is not a fragment*
5. $\{(b, (v_1, x))\} :: \vec{m} \equiv_\perp \{(b, (v_2, x))\} :: \vec{m}$ *provided* x *is not a fragment*
6. $m_2 :: \vec{m} \equiv_\perp m_1 :: (m_2 \setminus m_1) :: \vec{m}$ *provided that* $m_1 \subseteq m_2$
7. $m :: \vec{m} \leq_\perp \mathsf{lock}\ b\ m :: \vec{m}$ *provided that* $\mathsf{perm}\ b\ m = x$ *for some* x *that is not a fragment*
8. $m :: \vec{m} \leq_\perp \mathsf{unlock}\ \Omega\ m :: \vec{m}$

Here, a permission x *is a* fragment *if there is a* y *such that* $x \perp y$.

3. The language

In this section we define the syntax of expressions and statements.

DEFINITION 3.1. Expressions *are defined as:*

$$\circledcirc \in \mathsf{binop} ::= \texttt{==} \mid \texttt{<=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \ldots$$

$$e \in \mathsf{expr} ::= x_i \mid [v]_\Omega \mid e_1 := e_2 \mid f(\vec{e}) \mid \texttt{load}\ e \mid \texttt{alloc} \mid \texttt{free}\ e \mid e_1 \circledcirc e_2 \mid e_1\ ?\ e_2 : e_3 \mid (\tau)\,e$$

Instead of $[v]_\emptyset$*, we just write* v.

Expressions may contain side-effects: assignments $e_1 := e_2$, function calls $f(\vec{e})$, and allocation `alloc` and deallocation `free` e of dynamic memory. Unary operators, prefix and postfix increment

and decrement, assignment operators, and the comma operator, are omitted in this paper, but are included in the Coq formalization. The logical operators are defined in terms of the conditional.

Values $[v]_\Omega$ are annotated with a finite set Ω of *locked* memory indices. This set is initially empty, but whenever a write is performed, the written object is locked in memory and its memory index is added to Ω (see Section 4). Whenever a sequence point occurs, the locked objects in Ω will be unlocked in memory. The operation $\mathsf{locks}\ e$ collects the annotated locks in e.

Stacks ($\rho \in \mathsf{stack}$) are lists of memory indices. Variables are De Bruijn indices, *i.e.* the variable x_i refers to the ith memory index on the stack. De Bruijn indices avoid us from having to deal with shadowing due to block scope variables. Especially in the axiomatic semantics this is useful, as we do not want to lose information by a local variable shadowing an already existing one. The stack contains references to the value of each variable instead of the values itself so as to treat pointers to both local and allocated storage in a uniform way. Execution of a variable x_i thus consists of looking up its address b at position i in the stack, and returning a pointer $\mathsf{ptr}\, b$ to that address. Execution of `load` e consists of evaluating e to $\mathsf{ptr}\, b$ and looking up b in the memory.

DEFINITION 3.2. Statements *are defined as:*

$$s \in \mathsf{stmt} ::= e \mid \texttt{skip} \mid \texttt{goto}\ l \mid \texttt{return}\ e \mid \texttt{block}\ c\ s \mid s_1\,;s_2 \mid l : s \mid \texttt{while}(e)\ s \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2$$

The statement syntax is adapted from Krebbers and Wiedijk [21], but we treat assignments and function calls as expression constructs instead of statements constructs. Hence, assignments and function calls can be nested, and can occur in the expressions of a return, while, and conditional statement.

The construct `block` $c\ s$ opens a block scope with one local variable, where the boolean c specifies whether the variable is `const`-qualified or not. The permission $\mathsf{blockperm}\ c$ is defined as: $\mathsf{blockperm\ True} := \mathsf{R}$ and $\mathsf{blockperm\ False} := \mathsf{W}$. Since we use De Bruijn indices, the construct `block` $c\ s$ is nameless.

4. Operational semantics

We define the semantics of expressions and statements by a small step operational semantics. That means, computation is defined by the reflexive transitive closure of a reduction relation $\rightarrow$ on *program states*. To define this reduction, we first define head reduction of expressions, and then use *evaluation contexts* (as introduced by Felleisen *et al.* [14]) to define reduction of whole programs. In the remainder of this paper, we will use a memory instantiated with full permissions (Definition 2.6).

DEFINITION 4.1. *Given a stack ρ,* head reduction *of expressions $(e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$ is inductively defined as:*

1. $(x_i, m) \rightarrow_{\mathsf{h}} (\mathsf{ptr}\, b, m)$, *in case* $\rho\, i = b$
2. $([\mathsf{ptr}\, b]_{\Omega_1} := [v]_{\Omega_2}, m) \rightarrow_{\mathsf{h}} ([v]_{\{b\}\cup\Omega_1\cup\Omega_2}, \mathsf{lock}\ b\ (m[b := v]))$, *in case* $\mathsf{perm}\ b\ m = \gamma$ *and* $\mathsf{Write} \subseteq \mathsf{kind}\ \gamma$
3. $(\texttt{load}\ [\mathsf{ptr}\, b]_\Omega, m) \rightarrow_{\mathsf{h}} ([v]_\Omega, m)$, *for any v with* $m\ !!\ b = v$
4. $(\texttt{alloc}, m) \rightarrow_{\mathsf{h}} (\mathsf{ptr}\, b, \mathsf{alloc}\ b\ \mathsf{indet}\ \mathsf{F}\ m)$, *for any b with* $\mathsf{perm}\ b\ m = \bot$
5. $(\texttt{free}\ [\mathsf{ptr}\, b]_\Omega, m) \rightarrow_{\mathsf{h}} ([\mathsf{indet}]_\Omega, \mathsf{free}\ b\ m)$, *in case* $\mathsf{perm}\ b\ m = \gamma$ *and* $\mathsf{kind}\ \gamma = \mathsf{Free}$
6. $([v_1]_{\Omega_1} \circledcirc [v_2]_{\Omega_2}, m) \rightarrow_{\mathsf{h}} ([v']_{\Omega_1\cup\Omega_2}, m)$, *in case* $v_1 \circledcirc v_2 = v'$
7. $([v]_\Omega\ ?\ e_2 : e_3, m) \rightarrow_{\mathsf{h}} (e_2, \mathsf{unlock}\ \Omega\ m)$, *in case* $\mathsf{istrue}\ v$
8. $([v]_\Omega\ ?\ e_2 : e_3, m) \rightarrow_{\mathsf{h}} (e_3, \mathsf{unlock}\ \Omega\ m)$, *in case* $\mathsf{isfalse}\ v$
9. $((\tau)\,[v]_\Omega, m) \rightarrow_{\mathsf{h}} ([v']_\Omega, m)$, *in case* $(\tau)\, v = v'$

If the stack ρ is not clear from the context, we write $\rho \vdash (e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$. Note that picking an unused index for allocation in Rule 4 is non-deterministic.

In Rules 6 and 9, $v_1 \circledcirc v_2$ and $(\tau)\, v$ are partial functions that evaluate a binary operation on values and integer cast on values respectively. These functions fail in case of integer overflow, or if an operation is used wrongly (*e.g.* division by zero).

An assignment $[\mathsf{ptr}\, b]_{\Omega_1} := [v]_{\Omega_2}$ (Rule 2) not only stores the value v at index b, but also locks b in the memory. Locking b enforces the sequence point restriction, because consecutive reads and writes to b will fail (as ensured by the side-condition of the assignment rule, and Definition 2.8 of the $\_\ !!\ \_$ operation). In order to keep track of the lock of b, we add b to the set $\{b\}\cup\Omega_1\cup\Omega_2$. Rules 7 and 8 for the conditional $[v]_\Omega\ ?\ e_2 : e_3$ model a sequence point by unlocking the indices Ω in the memory, making future reads and writes possible again. Other constructs with a sequence point will be given a similar semantics (see Definition 4.10).

Like Ellison and Rosu [12], we implicitly use non-determinism to capture undefined behavior due to sequence point violations. For example, in `x + (x = 10)` only one execution order (performing the read after the assignment) leads to a sequence point violation. In Norrish's semantics [25] both execution orders lead to a sequence point violation as he also keeps track of reads.

Our treatment of sequence points assigns undefined behavior to more programs than the C standard, and Ellison and Rosu do. Ellison and Rosu release the locks of all objects at a sequence point, whereas we just release the locks that have been created by the subexpression where the sequence point occurred. For example, we assign undefined behavior to the following program of Ellison.

```
int x, y, *p = &y;
int f() { if (x) { p = &x; } return 0; }
int main() {
  return (x = 1) + (*p = 2) + f();
}
```

The execution order that leads to undefined behavior is (a) `x = 1`, (b) call `f` which changes `p` to `&x`, (c) `*p = 2`. Here, the lock of `&x` survives the function call. In the semantics of Ellison and Rosu, this program has defined behavior, as the sequence point before the function call releases all locks, so also the lock of `&x`.

We believe that this is a reasonable trade-off, because dealing with sequence points *locally* instead of *globally* brings the operational semantics closer to the axiomatic semantics as separation logic only talks about a local part of the memory. We believe only artificial programs become illegal, because different function calls in the same expression can still write to a shared part of the memory (which is useful for memoization). For example, given

```
int f(int y) {
  static int map[MAP_SIZE];
  if (map[y]) { return map[y]; }
  return map[y] = expensive_function(y);
}
```

the expression `f(3) + f(3)` has defined behavior according to our semantics of sequence points.

Since the C standard does not allow interleaved execution of function calls [16, 6.5.2.2p10], these are not described by the head reduction $\rightarrow_{\mathsf{h}}$. Instead, a function call changes the whole program state to a state in which its body is executed. When execution of the function body is finished, the result will be plugged back into the whole expression. To describe this behavior, and to select a head redex in an expression, we define *expression contexts*.

DEFINITION 4.2. Singular expression contexts *are defined as:*

$$\mathcal{E}_{\mathsf{s}} \in \mathsf{ectx_s} ::= \square := e \mid e := \square \mid f(\vec{e}_1, \square, \vec{e}_2) \mid \texttt{load}\ \square \mid \texttt{free}\ \square \mid \square \circledcirc e_2 \mid e_1 \circledcirc \square \mid \square\ ?\ e_2 : e_3 \mid (\tau)\,\square$$

Expression contexts *($\mathcal{E} \in \mathsf{ectx}$) are lists of singular contexts. Given an expression context $\mathcal{E}$ and an expression e, the* substitution of e for $\square$ in $\mathcal{E}$*, notation $\mathcal{E}[\,e\,]$, is defined as usual.*

In the reduction of whole programs (Definition 4.10), we allow $\mathcal{E}[\,e_1\,]$ to reduce to $\mathcal{E}[\,e_2\,]$ provided that $(e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$. To enforce that the first operand of the conditional $e_1 \ ?\ e_2 : e_3$ is executed entirely before the others, it is essential that we omit the contexts $e_1 \ ?\ \square : e_3$ and $e_1 \ ?\ e_2 : \square$.

The reduction of whole programs uses a *zipper*-like data structure, called a *program context* [21], to store the location of the substatement that is being executed. Execution of the program occurs by traversal through the program context in the direction *down* $\searrow$, *up* $\nearrow$, *jump* $\curvearrowright$, or *top* $\Uparrow$. When a `goto` l statement is executed, the direction is changed to $\curvearrowright l$, and the semantics performs a small step traversal through the program context until the label l has been reached. Program contexts extend the zipper by annotating each block scope variable with its associated memory index, and furthermore contain the full call stack of the program.

Program contexts can also be seen as a generalization of continuations (as for example being used in CompCert [2, 23]). However, there are some notable differences.

- Program contexts implicitly contain the stack, whereas a continuation semantics typically stores the stack separately.
- Program contexts also contain the part of the program that has been executed, whereas continuations only contain the part that remains to be done.
- Since the complete program is preserved, looping constructs like the while statement do not have to duplicate code.

The fact that program contexts do not throw away the parts of the statement that have been executed is essential for the treatment of goto. Upon an invocation of a goto, the semantics traverses through the program context until the corresponding label has been found. During this traversal it passes all block scope variables that go out of scope, allowing it to perform required allocations and deallocations in a natural way. Hence, the point of this traversal is not so much to *search* for the label, but much more to incrementally *calculate* the required allocations and deallocations.

DEFINITION 4.3. Singular statement contexts *are defined as:*

$$\begin{aligned}\mathcal{S}_{\mathsf{s}} \in \mathsf{sctx_s} ::= {} & \square\,;s_2 \mid s_1\,;\square \mid l:\square \mid \texttt{while}(e)\ \square \\ & \mid \texttt{if}\ (e)\ \square\ \texttt{else}\ s_2 \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ \square\end{aligned}$$

Given a singular statement context $\mathcal{S}_{\mathsf{s}}$ and a statement s, substitution of s for $\square$ in $\mathcal{S}_{\mathsf{s}}$*, notation $\mathcal{S}_{\mathsf{s}}[\,s\,]$, is defined as usual.*

A pair $(\vec{\mathcal{S}}_{\mathsf{s}},\ s)$ consisting of a list $\vec{\mathcal{S}}_{\mathsf{s}}$ of singular statement contexts and a statement s forms a zipper for statements without block scope variables. That means, $\vec{\mathcal{S}}_{\mathsf{s}}$ is a statement turned inside-out that represents a path from the focused substatement s to the top of the whole statement.

DEFINITION 4.4. Expression statement contexts *and* singular program contexts *are defined as:*

$$\mathcal{S}_{\mathsf{e}} \in \mathsf{sctx_e} ::= \square \mid \texttt{return}\ \square \mid \texttt{while}(\square)\ s \mid \texttt{if}\ (\square)\ s_1\ \texttt{else}\ s_2$$

$$\mathcal{P}_{\mathsf{s}} \in \mathsf{ctx_s} ::= \mathcal{S}_{\mathsf{s}} \mid \texttt{block}_b\ c\ \square \mid (\mathcal{S}_{\mathsf{e}}, e) \mid \mathsf{resume}\ \mathcal{E} \mid \mathsf{params}\ \vec{b}$$

Program contexts *($k \in \mathsf{ctx}$) are lists of singular program contexts. Given an expression statement context $\mathcal{S}_{\mathsf{e}}$ and an expression e,* substitution of e for $\square$ in $\mathcal{S}_{\mathsf{e}}$*, notation $\mathcal{S}_{\mathsf{e}}[\,e\,]$, is defined as usual.*

The previously defined program contexts will be used as follows in the operational semantics.

- When entering a block scope `block` $c\ s$, the singular context $\texttt{block}_b\ c\ \square$ is appended to the head of the program context. It associates the block scope with its memory index b.
- When executing a statement construct $\mathcal{S}_{\mathsf{e}}[\,e\,]$ that contains an expression e, the singular context $(\mathcal{S}_{\mathsf{e}}, e)$ is appended to the head of the program context to keep track of the statement. We need to keep track of the expression e as well so that it can be restored when execution of the expression is finished.
- When executing a function call $\mathcal{E}[\,f(\vec{v})\,]$, the singular context $\mathsf{resume}\ \mathcal{E}$ is appended to the head of the program context. When the function returns with value v, execution of the expression $\mathcal{E}[\,v\,]$ with the return value v plugged in, is continued.
- When a function body is entered, the singular context $\mathsf{params}\ \vec{b}$ is appended to the head of the program context. It contains a list $\vec{b}$ of memory indices of the function parameters.

As program contexts implicitly contain the stack, we define a function to extract it from them.

DEFINITION 4.5. *The* corresponding stack $\mathsf{getstack}\,k$ *of a program context k is defined as:*

$$\begin{aligned}\mathsf{getstack}\ (\mathcal{S}_{\mathsf{s}} :: k) &:= \mathsf{getstack}\ k \\ \mathsf{getstack}\ (\texttt{block}_b\ c\ \square :: k) &:= b :: \mathsf{getstack}\ k \\ \mathsf{getstack}\ ((\mathcal{S}_{\mathsf{e}}, e) :: k) &:= \mathsf{getstack}\ k \\ \mathsf{getstack}\ (\mathsf{resume}\ \mathcal{E} :: k) &:= [\,] \\ \mathsf{getstack}\ (\mathsf{params}\ \vec{b} :: k) &:= \vec{b} +\!\!+ \mathsf{getstack}\ k\end{aligned}$$

We define $\mathsf{getstack}\ (\mathsf{resume}\ \mathcal{E} :: k)$ as $[\,]$ instead of $\mathsf{getstack}\ k$, as otherwise it would be possible to refer to the local variables of the calling function.

DEFINITION 4.6. Directions*,* focuses *and* program states *are defined as:*

$$\begin{aligned}d \in \mathsf{direction} &::= \searrow \mid \nearrow \mid \curvearrowright l \mid \Uparrow v \\ \phi \in \mathsf{focus} &::= (d,\ s) \mid e \mid \overline{\mathsf{call}}\ f\ \vec{v} \mid \overline{\mathsf{return}}\ v \mid \overline{\mathsf{undef}} \\ S \in \mathsf{state} &::= \mathbf{S}(k,\ \phi,\ m)\end{aligned}$$

A program state $\mathbf{S}(k,\ \phi,\ m)$ consists of a program context k, the part of the program ϕ that is focused, and the memory m. Similar to Leroy's Cmedium [23], we have five kinds of states:

- $(d,\ s)$ for execution of a statement s in direction d,
- e for execution of an expression e,
- $\overline{\mathsf{call}}\ f\ \vec{v}$ for calling a function f with arguments v,
- $\overline{\mathsf{return}}\ v$ for returning from a function with return value v, and
- $\overline{\mathsf{undef}}$ to capture undefined behavior.

We have additional states for execution of expressions and undefined behavior. The semantics of Leroy's Cminor [22], and the semantics of Krebbers and Wiedijk [21], do not have such states, because their expressions are deterministic and side-effect free. They capture undefined behavior by letting the reduction get stuck, whereas that will not work in the presence of non-determinism.

DEFINITION 4.7. *The relation* $\mathsf{allocparams}\ \gamma\ m_1\ \vec{b}\ \vec{v}\ m_2$ *allocates fresh blocks $\vec{b}$ for function parameters $\vec{v}$ with permission γ (non-deterministically). It is inductively defined as:*

1. $\mathsf{allocparams}\ \gamma\ m\ [\,]\ [\,]\ m$
2. *If* $\mathsf{allocparams}\ \gamma\ m_1\ \vec{b}\ \vec{v}\ m_2$ *and* $\mathsf{perm}\ b\ m_2 = \bot$*, then* $\mathsf{allocparams}\ \gamma\ m_1\ (b :: \vec{b})\ (v :: \vec{v})\ (\mathsf{alloc}\ b\ v\ \gamma\ m_2)$

DEFINITION 4.8. *An expression is a* redex *if it is of the following shape: (a)* x_i*, (b)* $[v_1]_{\Omega_1} := [v_2]_{\Omega_2}$*, (c)* $f([v_1]_{\Omega_1}, \ldots, [v_2]_{\Omega_2})$*, (d)* $\texttt{load}\ [v]_\Omega$*, (e)* $\texttt{alloc}$ *(f)* $\texttt{free}\ [v]_\Omega$*, (g)* $[v_1]_{\Omega_1} \circledcirc [v_2]_{\Omega_2}$*, (h)* $[v]_\Omega\ ?\ e_2 : e_3$*, or (i)* $(\tau)\,[v]_\Omega$*.*

DEFINITION 4.9. *An expression is* safe *in stack* ρ *and memory* m *if: (a) it is of the shape* $f(\vec{e})$*, or (b) there is an expression* e' *and memory* m' *such that* $\rho \vdash (e, m) \rightarrow_{\mathsf{h}} (e', m')$*.*

DEFINITION 4.10. *Given a finite partial function* δ *mapping function names to statements, the* small step reduction $S_1 \twoheadrightarrow S_2$ *is inductively defined as:*

1. For simple statements:
   - *(a)* $\mathbf{S}(k, (\searrow, \texttt{skip}), m) \twoheadrightarrow \mathbf{S}(k, (\nearrow, \texttt{skip}), m)$
   - *(b)* $\mathbf{S}(k, (\searrow, \texttt{goto}\ l), m) \twoheadrightarrow \mathbf{S}(k, (\curvearrowright l, \texttt{goto}\ l), m)$
   - *(c)* $\mathbf{S}(k, (\searrow, \mathcal{S}_{\mathsf{e}}[\,e\,]), m) \twoheadrightarrow \mathbf{S}((\mathcal{S}_{\mathsf{e}}, e) :: k, e, m)$
2. For expressions:
   - *(a)* $\mathbf{S}(k, \mathcal{E}[\,e_1\,], m_1) \twoheadrightarrow \mathbf{S}(k, \mathcal{E}[\,e_2\,], m_2)$ *for any* e_2 *and* m_2 *s.t.* $\mathsf{getstack}\ k \vdash (e_1, m_1) \rightarrow_{\mathsf{h}} (e_2, m_2)$
   - *(b)* $\mathbf{S}(k, (\searrow, \mathcal{E}[\,f([v_0]_{\Omega_0}, \ldots, [v_n]_{\Omega_n})\,]), m) \twoheadrightarrow \mathbf{S}(\mathsf{resume}\ \mathcal{E} :: k, \overline{\mathsf{call}}\ f\ \vec{v}, \mathsf{unlock}\ (\bigcup \vec{\Omega})\ m)$
   - *(c)* $\mathbf{S}(k, (\searrow, \mathcal{E}[\,e\,]), m) \twoheadrightarrow \mathbf{S}(k, \overline{\mathsf{undef}}, m)$, *provided that* e *is an unsafe redex*
3. For finished expressions:
   - *(a)* $\mathbf{S}((\square, e) :: k, [v]_\Omega, m) \twoheadrightarrow \mathbf{S}(k, (\nearrow, e), \mathsf{unlock}\ \Omega\ m)$
   - *(b)* $\mathbf{S}((\texttt{return}\ \square, e) :: k, [v]_\Omega, m) \twoheadrightarrow \mathbf{S}(k, (\Uparrow v, \texttt{return}\ e), \mathsf{unlock}\ \Omega\ m)$
   - *(c)* $\mathbf{S}((\texttt{while}(\square)\ s, e) :: k, [v]_\Omega, m) \twoheadrightarrow \mathbf{S}(\texttt{while}(e)\ \square :: k, (\searrow, s), \mathsf{unlock}\ \Omega\ m)$ *provided that* $\mathsf{istrue}\ v$
   - *(d)* $\mathbf{S}((\texttt{while}(\square)\ s, e) :: k, [v]_\Omega, m) \twoheadrightarrow \mathbf{S}(k, (\nearrow, \texttt{while}(e)\ s), \mathsf{unlock}\ \Omega\ m)$ *provided that* $\mathsf{isfalse}\ v$
   - *(e)* $\mathbf{S}((\texttt{if}\ (\square)\ s_1\ \texttt{else}\ s_2, e) :: k, [v]_\Omega, m) \twoheadrightarrow \mathbf{S}(\texttt{if}\ (e)\ \square\ \texttt{else}\ s_2 :: k, (\searrow, s_1), \mathsf{unlock}\ \Omega\ m)$ *provided that* $\mathsf{istrue}\ v$
   - *(f)* $\mathbf{S}((\texttt{if}\ (\square)\ s_1\ \texttt{else}\ s_2, e) :: k, [v]_\Omega, m) \twoheadrightarrow \mathbf{S}(\texttt{if}\ (e)\ s_1\ \texttt{else}\ \square :: k, (\searrow, s_2), \mathsf{unlock}\ \Omega\ m)$ *provided that* $\mathsf{isfalse}\ v$
4. For compound statements:
   - *(a)* $\mathbf{S}(k, (\searrow, \texttt{block}\ c\ s), m) \twoheadrightarrow \mathbf{S}((\texttt{block}_b\ c\ \square) :: k, (\searrow, s), \mathsf{alloc}\ b\ \mathsf{indet}\ (\mathsf{blockperm}\ c)\ m)$ *for any* b *such that* $\mathsf{perm}\ b\ m = \bot$
   - *(b)* $\mathbf{S}(k, (\searrow, s_1\,; s_2), m) \twoheadrightarrow \mathbf{S}((\square\,; s_2) :: k, (\searrow, s_1), m)$
   - *(c)* $\mathbf{S}(k, (\searrow, l : s), m) \twoheadrightarrow \mathbf{S}((l : \square) :: k, (\searrow, s), m)$
   - *(d)* $\mathbf{S}((\texttt{block}_b\ c\ \square) :: k, (\nearrow, s), m) \twoheadrightarrow \mathbf{S}(k, (\nearrow, \texttt{block}\ c\ s), \mathsf{free}\ b\ m)$
   - *(e)* $\mathbf{S}((\square\,; s_2) :: k, (\nearrow, s_1), m) \twoheadrightarrow \mathbf{S}((s_1\,; \square) :: k, (\searrow, s_2), m)$
   - *(f)* $\mathbf{S}((s_1\,; \square) :: k, (\nearrow, s_2), m) \twoheadrightarrow \mathbf{S}(k, (\nearrow, s_1\,; s_2), m)$
   - *(g)* $\mathbf{S}(\texttt{while}(e)\ \square :: k, (\nearrow, s), m) \twoheadrightarrow \mathbf{S}(k, (\searrow, \texttt{while}(e)\ s), m)$
   - *(h)* $\mathbf{S}((\texttt{if}\ (e)\ \square\ \texttt{else}\ s_2) :: k, (\nearrow, s_1), m) \twoheadrightarrow \mathbf{S}(k, (\nearrow, \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2), m)$
   - *(i)* $\mathbf{S}((\texttt{if}\ (e)\ s_1\ \texttt{else}\ \square) :: k, (\nearrow, s_2), m) \twoheadrightarrow \mathbf{S}(k, (\nearrow, \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2), m)$
   - *(j)* $\mathbf{S}((l : \square) :: k, (\nearrow, s), m) \twoheadrightarrow \mathbf{S}(k, (\nearrow, l : s), m)$
5. For function calls:
   - *(a)* $\mathbf{S}(k, \overline{\mathsf{call}}\ f\ \vec{v}, m_1) \twoheadrightarrow \mathbf{S}(\mathsf{params}\ \vec{b} :: k, (\searrow, s), m_2)$ *for any* s, $\vec{b}$ *and* m_2 *s.t.* $\delta\ f = s$ *and* $\mathsf{allocparams}\ \mathsf{W}\ m_1\ \vec{b}\ \vec{v}\ m_2$
   - *(b)* $\mathbf{S}(\mathsf{params}\ \vec{b} :: k, (\nearrow, s), m) \twoheadrightarrow \mathbf{S}(k, \overline{\mathsf{return}}\ \mathsf{indet}, \mathsf{free}\ \vec{b}\ m)$
   - *(c)* $\mathbf{S}(\mathsf{params}\ \vec{b} :: k, (\Uparrow v, s), m) \twoheadrightarrow \mathbf{S}(k, \overline{\mathsf{return}}\ v, \mathsf{free}\ \vec{b}\ m)$
   - *(d)* $\mathbf{S}(\mathsf{resume}\ \mathcal{E} :: k, \overline{\mathsf{return}}\ v, m) \twoheadrightarrow \mathbf{S}(k, \mathcal{E}[\,v\,], m)$
6. For non-local control flow:
   - *(a)* $\mathbf{S}((\texttt{block}_b\ c\ \square) :: k, (\Uparrow v, s), m) \twoheadrightarrow \mathbf{S}(k, (\Uparrow v, \texttt{block}\ c\ s), \mathsf{free}\ b\ m)$
   - *(b)* $\mathbf{S}(\mathcal{S}_{\mathsf{s}} :: k, (\Uparrow v, s), m) \twoheadrightarrow \mathbf{S}(k, (\Uparrow v, \mathcal{S}_{\mathsf{s}}[\,s\,]), m)$
   - *(c)* $\mathbf{S}(k, (\curvearrowright l, l : s), m) \twoheadrightarrow \mathbf{S}((l : \square) :: k, (\searrow, s), m)$
   - *(d)* $\mathbf{S}(k, (\curvearrowright l, \texttt{block}\ c\ s), m) \twoheadrightarrow \mathbf{S}((\texttt{block}_b\ c\ \square) :: k, (\curvearrowright l, s), \mathsf{alloc}\ b\ \mathsf{indet}\ (\mathsf{blockperm}\ c)\ m)$ *for any* b *such that* $\mathsf{perm}\ b\ m = \bot$*, and provided that* $l \in \mathsf{labels}\ s$
   - *(e)* $\mathbf{S}(\texttt{block}_b\ c\ \square :: k, (\curvearrowright l, s), m) \twoheadrightarrow \mathbf{S}(k, (\curvearrowright l, \texttt{block}\ c\ s), \mathsf{free}\ b\ m)$ *provided that* $l \notin \mathsf{labels}\ s$
   - *(f)* $\mathbf{S}(k, (\curvearrowright l, \mathcal{S}_{\mathsf{s}}[\,s\,]), m) \twoheadrightarrow \mathbf{S}(\mathcal{S}_{\mathsf{s}} :: k, (\curvearrowright l, s), m)$ *provided that* $l \in \mathsf{labels}\ s$
   - *(g)* $\mathbf{S}(k, (\curvearrowright l, s), m) \twoheadrightarrow \mathbf{S}(k, (\curvearrowright l, \mathcal{S}_{\mathsf{s}}[\,s\,]), m)$ *provided that* $l \notin \mathsf{labels}\ s$

Note that the selection of redexes in Rule 2a, 2b, and 2c is non-deterministic. Moreover, note that the rules 6c and 6f overlap, and that the splitting into $\mathcal{S}_{\mathsf{s}}$ *and* s *in rule 6f is non-deterministic.*

DEFINITION 4.11. *We let* $\twoheadrightarrow^*$ *denote the reflexive-transitive closure of* $\twoheadrightarrow$*, and let* $\twoheadrightarrow^n$ *denote paths of* $\leq n$ $\twoheadrightarrow$*-reduction steps.*

Execution of a statement $\mathbf{S}(k, (d, s), m)$ is performed by traversing through the program context k and statement s in direction d. The direction *down* $\searrow$ (respectively *up* $\nearrow$) is used to traverse downwards (respectively upwards) to the next substatement that has to be executed. When a substatement $\mathcal{S}_{\mathsf{e}}[\,e\,]$ containing an expression e has been reached (Rule 1c), the location of the expression $\mathcal{S}_{\mathsf{e}}$ is stored on the program context, and execution is continued in $\mathbf{S}((\mathcal{S}_{\mathsf{e}}, e) :: k, e, m)$.

Execution of an expression $\mathbf{S}(k, e, m)$ is performed by non-deterministically decomposing e into $\mathcal{E}[\,e'\,]$. Rule 2a allows e' to perform a $\rightarrow_{\mathsf{h}}$-step, and Rule 2b allows e' to perform a function call. If the redex e' is unsafe and thereby cannot be contracted (*e.g.* because of a sequence violation or integer overflow), Rule 2c ensures that the whole program reduces to the $\overline{\mathsf{undef}}$ state. When execution of an expression has resulted in a value $[v]_\Omega$, we model a sequence point by unlocking Ω in memory (Rule 3a-3f).

For a function call $\mathbf{S}(k, \mathcal{E}[\,f([v_0]_{\Omega_0}, \ldots, [v_n]_{\Omega_n})\,], m)$, two reductions occur before the function body will be executed. The first to $\mathbf{S}(\mathsf{resume}\ \mathcal{E} :: k, \overline{\mathsf{call}}\ f\ \vec{v}, \mathsf{unlock}\ (\bigcup \vec{\Omega})\ m)$ (Rule 2b) stores the location of the caller on the program context and takes care of the sequence point before the function call. The subsequent reduction to $\mathbf{S}(\mathsf{params}\ \vec{b} :: \mathsf{resume}\ \mathcal{E} :: k, (\searrow, s), m')$ (Rule 5a) looks up the callee's body s, allocates the parameters $\vec{v}$, and then performs a transition to execute the function body s.

We consider two directions for non-local control flow: *jump* $\curvearrowright l$ and *top* $\Uparrow v$. After a $\texttt{goto}\ l$ (Rule 1b), the direction $\curvearrowright l$ is used to traverse to the substatement labeled l (Rule 6c-6g). Although this traversal is non-deterministic (in the case of duplicate labels), there are some side-conditions in order to ensure that the reduction is not going back and forth between the same locations. This is required because we may otherwise impose non-terminating behavior on terminating programs. The non-determinism could be removed entirely by adding additional side-conditions. However, as we already have other sources of non-determinism, we omitted doing so to ease formalization.

When execution of the expression e of a $\texttt{return}\ e$ statement has resulted in a value v (Rule 3b), the direction $\Uparrow v$ is used to traverse to the *top* of the whole statement (Rule 6a and 6b). When this traversal reaches the top of the statement, there are two reductions to give the return value v to the caller. The first reduction, from $\mathbf{S}(\mathsf{params}\ \vec{b} :: \mathsf{resume}\ \mathcal{E} :: k, (\Uparrow v, s), m)$ to $\mathbf{S}(\mathsf{resume}\ \mathcal{E} :: k, \overline{\mathsf{return}}\ v, \mathsf{free}\ \vec{b}\ m)$ (Rule 5c), deallocates the function parameters, and the second reduction, to $\mathbf{S}(k, \mathcal{E}[\,v\,], \mathsf{free}\ \vec{b}\ m)$ (Rule 5d), resumes execution of the expression $\mathcal{E}[\,v\,]$ at the caller.

5. Axiomatic semantics

Judgments of Hoare logic are triples $\{P\}\, s\, \{Q\}$, where s is a statement, and P and Q are *assertions* called the pre- and postcondition. The intuitive reading of such a triple is: if P holds for the memory before executing s, and execution of s terminates, then Q holds afterwards. For our language, we have two such judgments: one for expressions and one for statements.

Our expression judgments are quadruples $\Delta \vdash \{P\}\, e\, \{Q\}$. As usual, P and Q are the pre- and postcondition of e respectively, but whereas P is just an assertion, Q is a function from values to assertions. It ensures that if execution of e yields a value v, then $Q\, v$ holds afterwards. The environment Δ is a finite function from function names to their pre- and postconditions that is used to cope with (mutually) recursive functions.

As in Krebbers and Wiedijk [21], our judgments for statements are sextuples $\Delta;\ R;\ J \vdash \{P\}\, s\, \{Q\}$, where R is a function from values to assertions, and J is a function from labels to assertions. The assertion $R\, v$ has to hold when executing a `return` e (for each value v obtained by execution of e), and $J\, l$ is the jumping condition that has to hold when executing a `goto` l.

We use a shallow embedding to represent assertions. This treatment is similar to that of Appel and Blazy [2], Von Oheimb [29], Krebbers and Wiedijk [21], *etc.* In order to talk about expressions without side-effects (assignments, allocation and deallocation, and function calls) in assertions (we need this for various rules of our axiomatic semantics, see Definition 5.9), we define an evaluation function for *pure* expressions.

DEFINITION 5.1. Evaluation $[\![\, e \,]\!]_{\rho,m}$ of an expression e *in a stack* ρ *and memory* m *is a partial function that is defined as:*

$$
\begin{aligned}
[\![\, x_i \,]\!]_{\rho,m} &:= \mathsf{ptr}\, b && \text{if } \rho\, i = b\\
[\![\, v \,]\!]_{\rho,m} &:= v\\
[\![\, \texttt{load}\ e \,]\!]_{\rho,m} &:= m\ !!\ b && \text{if } [\![\, e \,]\!]_{\rho,m} = \mathsf{ptr}\, b\\
[\![\, e_1 \circledcirc e_2 \,]\!]_{\rho,m} &:= [\![\, e_1 \,]\!]_{\rho,m} \circledcirc [\![\, e_2 \,]\!]_{\rho,m}\\
[\![\, e_1\ ?\ e_2 : e_3 \,]\!]_{\rho,m} &:= \begin{cases} [\![\, e_2 \,]\!]_{\rho,m} & \text{if } [\![\, e_1 \,]\!]_{\rho,m} = v \text{ and } \mathsf{istrue}\ v\\ [\![\, e_3 \,]\!]_{\rho,m} & \text{if } [\![\, e_1 \,]\!]_{\rho,m} = v \text{ and } \mathsf{isfalse}\ v \end{cases}\\
[\![\, (\tau)\, e \,]\!]_{\rho,m} &:= (\tau)\, [\![\, e \,]\!]_{\rho,m}
\end{aligned}
$$

DEFINITION 5.2. Assertions *are predicates over the the stack and the memory. We define the following connectives on assertions.*

$$
\begin{aligned}
P \to Q &:= \lambda\rho\, m\,.\, P\, \rho\, m \to Q\, \rho\, m & \forall x\,.\, P\, x &:= \lambda\rho\, m\,.\, \forall x\,.\, P\, x\, \rho\, m\\
P \wedge Q &:= \lambda\rho\, m\,.\, P\, \rho\, m \wedge Q\, \rho\, m & \exists x\,.\, P\, x &:= \lambda\rho\, m\,.\, \exists x\,.\, P\, x\, \rho\, m\\
P \vee Q &:= \lambda\rho\, m\,.\, P\, \rho\, m \vee Q\, \rho\, m & \ulcorner P \urcorner &:= \lambda\rho\, m\,.\, P\\
\neg P &:= \lambda\rho\, m\,.\, \neg P\, \rho\, m & e \Downarrow v &:= \lambda\rho\, m\,.\, [\![\, e \,]\!]_{\rho,m} = v
\end{aligned}
$$

We treat $\ulcorner \_ \urcorner$ *as an implicit coercion, e.g. we write* True *instead of* $\ulcorner \mathsf{True} \urcorner$*. Also, we often lift these connectives to functions to assertions, e.g. we write* $P \wedge Q$ *instead of* $\lambda v\,.\, P\, v \wedge Q\, v$*.*

DEFINITION 5.3. *We let* $P \vDash Q$ *denote that for all stacks* ρ *and memories* m *we have* $P\, \rho\, m$ *implies* $Q\, \rho\, m$*.*

DEFINITION 5.4. *An assertion* P *is called* stack independent *if for all* $\rho_1, \rho_2 \in \mathsf{stack}$ *and* $m \in \mathsf{mem}$ *with* $P\, \rho_1\, m$ *we have* $P\, \rho_2\, m$*. Similarly,* P *is called* unlock independent *if for all* $\rho \in \mathsf{stack}$*,* $m \in \mathsf{mem}$ *and* $\Omega \subseteq \mathsf{index}$ *with* $P\, \rho\, m$ *we have* $P\, \rho\, (\mathsf{unlock}\ \Omega\ m)$*.*

Next, we define the assertions of separation logic [28]. The *separating conjunction* $P * Q$ asserts that the memory can be split into two disjoint parts such that P holds in the one part, and Q in the other (due to the use of fractional permissions, these parts may overlap as long as their permissions are disjoint and their values agree). Finally, $e_1 \overset{\gamma}{\mapsto} e_2$ asserts that the memory consists of exactly one object at e_1 with contents e_2 and permission γ.

DEFINITION 5.5. *The* connectives of separation logic *are:*

$$
\begin{aligned}
\mathsf{emp} &:= \lambda\rho\, m\,.\, m = \emptyset\\
P * Q &:= \lambda\rho\, m\,.\, \exists m_1\, m_2\,.\, m = m_1 \cup m_2 \wedge\\
&\qquad m_1 \perp m_2 \wedge P\, \rho\, m_1 \wedge Q\, \rho\, m_2\\
e_1 \overset{\gamma}{\mapsto} e_2 &:= \lambda\rho\, m\,.\, \exists b\, v\,.\, [\![\, e_1 \,]\!]_{\rho,m} = \mathsf{ptr}\, b \wedge\\
&\qquad [\![\, e_2 \,]\!]_{\rho,m} = v \wedge m = \{(b, (v, \gamma))\}\\
e_1 \overset{\gamma}{\mapsto} - &:= \exists e_2\,.\, e_1 \overset{\gamma}{\mapsto} e_2
\end{aligned}
$$

To enforce the sequence point restriction, the rule for assignment changes the permission γ of $e_1 \overset{\gamma}{\mapsto} e_2$ into $\mathsf{lock}\ \gamma$. Subsequent reads and writes are therefore no longer possible. At constructs that have a sequence point, we use the assertion $P \triangleright$ to release these locks, and to make future reads and writes possible again.

DEFINITION 5.6. *The* unlocking assertion $P \triangleright$ *is defined as:*

$$P \triangleright := \lambda\rho\, m\,.\, P\, \rho\, (\mathsf{unlock}\ (\mathsf{locks}\ m)\ m).$$

We proved properties as $P \triangleright * Q \triangleright \vDash (P * Q) \triangleright$ to push the $\triangleright$-connective through an assertion.

Similar to Krebbers and Wiedijk [21], we need to lift an assertion such that the De Bruijn indices of its variables are increased so as to deal with block scope variables. The lifting $P \uparrow$ of P is defined semantically, and we prove that it behaves as expected.

DEFINITION 5.7. *The* lifting assertion $P \uparrow$ *is defined as:*

$$P \uparrow := \lambda\rho\, m\,.\, P\, (\mathsf{tail}\, \rho)\, m.$$

LEMMA 5.8. *The operation* $(\_) \uparrow$ *distributes over the connectives* $\to$*,* $\wedge$*,* $\vee$*,* $\neg$*,* $\forall$*,* $\exists$*, and* $*$*. We have* $(e \Downarrow v) \uparrow\ =\ (e \uparrow) \Downarrow v$ *and* $(e_1 \overset{\gamma}{\mapsto} e_2) \uparrow = (e_1 \uparrow) \overset{\gamma}{\mapsto} (e_2 \uparrow)$*, where the operation* $e \uparrow$ *replaces each variable* x_i *in* e *by* x_{i+1}*.*

The specification of a function with parameters $\vec{v}$ consists of an assertion $P\, \vec{y}\, \vec{v}$ called the precondition, and a function $Q\, \vec{y}\, \vec{v}$ from (return) values to assertions called the postcondition. We allow universal quantification over arbitrary logical variables $\vec{y}$ in order to relate the pre- and postcondition. The pre- and postcondition should moreover be stack independent because local variables will have a different meaning at the caller than at callee. We denote such a specification as $\forall \vec{y} \forall \vec{v}\,.\, \{P\, \vec{y}\, \vec{v}\}\ \{Q\, \vec{y}\, \vec{v}\}$.

DEFINITION 5.9. *Given a finite partial function* δ *mapping function names to statements, the expression judgment* $\Delta \vdash \{P\}\, e\, \{Q\}$ *and statement judgment* $\Delta;\ R;\ J \vdash \{P\}\, s\, \{Q\}$ *of* the axiomatic semantics *are mutually inductively defined as shown in Figure 1.*

We have a frame, weaken, and exist rule for both the expression and statement judgments. The traditional frame rule of separation logic [28] includes a side-condition $\mathsf{modifies}\ s \cap \mathsf{free}\ A = \emptyset$. Like Krebbers and Wiedijk [21], we do not need this side-condition as our local variables are (immutable) references into the memory. Since the return and goto statements leave the normal control flow, the postconditions of the (goto) and (return) rules are arbitrary.

The rules for function calls are based on those by Krebbers and Wiedijk [21]. The (e-call) rule is used to call a function $f(\vec{e})$ that is already in Δ. The assertion $B\, \vec{v}$ is used to frame a part of the memory that is used by $\vec{e}$ but not by the f itself.

The (add funs) rule can be used to add an arbitrary family Δ' of specifications of (possibly mutually recursive) functions to Δ. For each function f in Δ' with precondition P' and postcondition Q', it has to be verified that the function body $\delta\, f$ is correct for all instantiations of the logical variables $\vec{y}$ and input values $\vec{v}$. The precondition $\Pi_*[\, x_i \overset{\mathsf{W}}{\mapsto} v_i \,] * P'\, \vec{y}\, \vec{v}$, where $\Pi_*[\, e_i \overset{\gamma_i}{\mapsto} e_i' \,]$ denotes the

$$\frac{\Delta \vdash \{P\}\, e\, \{Q\}}{\Delta \vdash \{A * P\}\, e\, \{A * Q\}}\ \text{(e-frame)} \qquad \frac{\forall x\,.\,(\Delta \vdash \{P\,x\}\, e\, \{Q\})}{\Delta \vdash \{\exists x\,.\,P\,x\}\, e\, \{Q\}}\ \text{(e-exists)} \qquad \frac{P' \vDash P \quad \Delta \vdash \{P\}\, e\, \{Q\} \quad (\forall v\,.\,Q\,v \vDash Q'\,v)}{\Delta \vdash \{P'\}\, e\, \{Q'\}}\ \text{(e-weaken)}$$

$$\frac{P \vDash e \Downarrow v}{\Delta \vdash \{P\}\, e\, \{\lambda v'\,.\,v = v' \wedge P\}}\ \text{(e-base)} \qquad \frac{}{\Delta \vdash \{\mathsf{emp}\}\, \mathtt{alloc}\, \{\lambda a\,.\,a \overset{\mathsf{F}}{\mapsto} -\}}\ \text{(e-alloc)} \qquad \frac{\Delta \vdash \{P\}\, e\, \{\lambda a\,.\,Q * a \overset{\mathsf{F}}{\mapsto} -\}}{\Delta \vdash \{P\}\, \mathtt{free}\, e\, \{\lambda \_\,.\,Q\}}\ \text{(e-free)}$$

$$\frac{\mathsf{kind}\ \gamma \neq \mathsf{Locked} \quad \Delta \vdash \{P\}\, e\, \{\lambda a\,.\,\exists v\,.\,Q\,a\,v * a \overset{\gamma}{\mapsto} v\}}{\Delta \vdash \{P\}\, \mathtt{load}\, e\, \{\lambda v\,.\,\exists a\,.\,Q\,a\,v * a \overset{\gamma}{\mapsto} v\}}\ \text{(e-load)}$$

$$\frac{\mathsf{Write} \subseteq \mathsf{kind}\ \gamma \quad \Delta \vdash \{P_1\}\, e_1\, \{Q_1\} \quad \Delta \vdash \{P_2\}\, e_2\, \{Q_2\} \quad \forall a v\,.\,(Q_1\,a * Q_2\,v \vDash a \overset{\gamma}{\mapsto} - * R\,a\,v)}{\Delta \vdash \{P_1 * P_2\}\, e_1 := e_2\, \{\lambda v\,.\,\exists a\,.\,a \overset{\gamma}{\mapsto} v * R\,a\,v\}}\ \text{(e-assign)}$$

$$\frac{\Delta \vdash \{P_1\}\, e_1\, \{Q_1\} \quad \Delta \vdash \{P_2\}\, e_2\, \{Q_2\} \quad \forall v_1\,v_2\,.\,(Q_1\,v_1 * Q_2\,v_2 \vDash \exists v\,.\,R\,v\,v_1\,v_2 \wedge v_1 \circledcirc v_2 \Downarrow v)}{\Delta \vdash \{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{\lambda v\,.\,\exists v_1\,v_2\,.\,R\,v\,v_1\,v_2\}}\ \text{(e-binop)}$$

$$\frac{\Delta\, f = (\forall \vec{z} \forall \vec{w}\,.\,\{P'\,\vec{z}\,\vec{w}\}\,\{Q'\,\vec{z}\,\vec{w}\}) \quad \forall i\,.\,(\Delta \vdash \{P_i\}\, e_i\, \{Q_i\}) \quad \forall \vec{v}\,.\,(\Pi_*[\,Q_i\,v_i\,] \vDash P'\,\vec{y}\,\vec{v} * A\,\vec{v}) \quad \forall \vec{v}\,v\,.\,(Q'\,\vec{y}\,\vec{v}\,v * A\,\vec{v} \vDash R\,v)}{\Delta \vdash \{\Pi_*[\,\vec{P}\,]\}\, f(\vec{e})\, \{R\}}\ \text{(e-call)}$$

$$\frac{\Delta \vdash \{P\}\, e_1\, \{\lambda v\,.\,v \neq \mathsf{indet} \wedge P'\,v \triangleright\} \quad \Delta \vdash \{\exists v\,.\,\mathsf{istrue}\ v \wedge P'\,v\}\, e_2\, \{Q\} \quad \Delta \vdash \{\exists v\,.\,\mathsf{isfalse}\ v \wedge P'\,v\}\, e_2\, \{Q\}}{\Delta \vdash \{P\}\, e_1\ ?\ e_2 : e_3\, \{Q\}}\ \text{(e-if)}$$

$$\frac{\Delta;\ R;\ J \vdash \{P\}\, s\, \{Q\}}{\Delta;\ A * R;\ A * J \vdash \{A * P\}\, s\, \{A * Q\}}\ \text{(frame)} \qquad \frac{\forall x\,.\,(\Delta;\ R;\ J \vdash \{P\,x\}\, s\, \{Q\})}{\Delta;\ R;\ J \vdash \{\exists x\,.\,P\,x\}\, s\, \{Q\}}\ \text{(exists)}$$

$$\frac{\forall v\,.\,(R\,v \vDash R'\,v) \quad \forall l \in \mathsf{labels}\ s\,.\,(J'\,l \vDash J\,l) \quad \forall l \notin \mathsf{labels}\ s\,.\,(J\,l \vDash J'\,l) \quad P' \vDash P \quad \Delta;\ R;\ J \vdash \{P\}\, s\, \{Q\} \quad Q \vDash Q'}{\Delta;\ R';\ J' \vdash \{P'\}\, s\, \{Q'\}}\ \text{(weaken)}$$

$$\frac{\Delta \vdash \{P\}\, e\, \{\lambda \_\,.\,Q \triangleright\}}{\Delta;\ R;\ J \vdash \{P\}\, e\, \{Q\}}\ \text{(expr)} \qquad \frac{}{\Delta;\ R;\ J \vdash \{P\}\, \mathtt{skip}\, \{P\}}\ \text{(skip)} \qquad \frac{\Delta;\ R;\ J \vdash \{P\}\, s_1\, \{P'\} \quad \Delta;\ R;\ J \vdash \{P'\}\, s_2\, \{Q\}}{\Delta;\ R;\ J \vdash \{P\}\, s_1\, ;\, s_2\, \{Q\}}\ \text{(comp)}$$

$$\frac{\Delta \vdash \{P\}\, e\, \{R \triangleright\}}{\Delta;\ R;\ J \vdash \{P\}\, \mathtt{return}\, e\, \{Q\}}\ \text{(return)} \qquad \frac{\Delta;\ R;\ J \vdash \{J\,l\}\, s\, \{Q\}}{\Delta;\ R;\ J \vdash \{J\,l\}\, l : s\, \{Q\}}\ \text{(label)} \qquad \frac{}{\Delta;\ R;\ J \vdash \{J\,l\}\, \mathtt{goto}\ l\, \{Q\}}\ \text{(goto)}$$

$$\frac{\gamma = \mathsf{blockperm}\ c \quad \Delta;\ x_0 \overset{\gamma}{\mapsto} - * J\uparrow;\ x_0 \overset{\gamma}{\mapsto} - * R\uparrow \vdash \{x_0 \overset{\gamma}{\mapsto} - * P\uparrow\}\, s\, \{x_0 \overset{\gamma}{\mapsto} - * Q\uparrow\}}{\Delta;\ R;\ J \vdash \{P\}\, \mathtt{block}\ c\, s\, \{Q\}}\ \text{(block)}$$

$$\frac{\Delta \vdash \{P\}\, e\, \{\lambda v\,.\,v \neq \mathsf{indet} \wedge Q\,v \triangleright\} \quad \Delta;\ R;\ J \vdash \{\exists v\,.\,\mathsf{istrue}\ v \wedge Q\,v\}\, s\, \{P\}}{\Delta;\ R;\ J \vdash \{P\}\, \mathtt{while}(e)\ s\, \{\exists v\,.\,\mathsf{isfalse}\ v \wedge Q\,v\}}\ \text{(while)}$$

$$\frac{\Delta \vdash \{P\}\, e\, \{\lambda v\,.\,v \neq \mathsf{indet} \wedge P'\,v \triangleright\} \quad \Delta;\ R;\ J \vdash \{\exists v\,.\,\mathsf{istrue}\ v \wedge P'\,v\}\, s_1\, \{Q\} \quad \Delta;\ R;\ J \vdash \{\exists v\,.\,\mathsf{isfalse}\ v \wedge P'\,v\}\, s_1\, \{Q\}}{\Delta;\ R;\ J \vdash \{P\}\, \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2\, \{Q\}}\ \text{(if)}$$

$$\frac{\begin{array}{l}\forall f\,P'\,Q'\,.\,\Delta' f = (\forall \vec{z} \forall \vec{w}\,.\,\{P'\,\vec{z}\,\vec{w}\}\,\{Q'\,\vec{z}\,\vec{w}\}) \to \forall \vec{y}\,\vec{v} \\ \qquad (\Delta' \cup \Delta;\ \lambda l\,.\,\mathsf{False};\ \lambda v\,.\,\Pi_*[\,x_i \overset{\mathsf{W}}{\mapsto} -\,] * Q'\,\vec{y}\,\vec{v}\,v \vdash \{\Pi_*[\,x_i \overset{\mathsf{W}}{\mapsto} v_i\,] * P'\,\vec{y}\,\vec{v}\}\,\delta\, f\, \{\Pi_*[\,x_i \overset{\mathsf{W}}{\mapsto} -\,] * Q'\,\vec{y}\,\vec{v}\,\mathsf{indet}\}) \\ \Delta' \cup \Delta;\ R;\ J \vdash \{P\}\, s\, \{Q\} \qquad\qquad \mathsf{dom}\ \Delta' \subseteq \mathsf{dom}\ \delta\end{array}}{\Delta;\ R;\ J \vdash \{P\}\, s\, \{Q\}}\ \text{(add funs)}$$

Figure 1. The rules of the axiomatic semantics.

assertion $e_0 \overset{\gamma_0}{\mapsto} e_0' * \cdots * e_n \overset{\gamma_n}{\mapsto} e_n'$, states that the function parameters $\vec{x}$ are allocated with values $\vec{v}$ for which the precondition P' of the function holds. The postcondition $\Pi_*[\,x_i \overset{\mathsf{W}}{\mapsto} -\,] * Q'\vec{y}\,\vec{v}\,\mathsf{indet}$ and returning condition $\lambda v\,.\,\Pi_*[\,x_i \overset{\mathsf{W}}{\mapsto} -\,] * Q'\vec{y}\,\vec{v}\,v$ ensure that the parameters have not been deallocated during the execution of the function and that the postcondition Q' holds for the return value. The jumping condition $\lambda l\,.\,\mathsf{False}$ ensures that all gotos jump to a label that occurs in the function body.

Our axiomatic semantics is at least as powerful as an ordinary separation logic for C because not only variants of the ordinary inference rules can be derived, but also derived rules for more complex constructs. For example

$$\frac{e_1 \overset{\gamma_1}{\mapsto} - * e_2 \overset{\gamma_2}{\mapsto} - * P \vDash e_3 \Downarrow v}{\{e_1 \overset{\gamma_1}{\mapsto} - * e_2 \overset{\gamma_2}{\mapsto} - * P\}\, e_1 := e_2 := e_3\, \{e_1 \overset{\gamma_1}{\mapsto} v * e_2 \overset{\gamma_2}{\mapsto} v * P\}}$$

provided that e_1 and e_2 are `load`-free, P is unlock independent and $\mathsf{Write} \subseteq \mathsf{kind}\ \gamma_1, \mathsf{kind}\ \gamma_2$.

6. Soundness of the axiomatic semantics

We will define judgments $\Delta \vDash \{P\}\, e\, \{Q\}$ (Definition 6.6) and $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$ (Definition 6.7) to describe partial program correctness. The judgment $\Delta \vDash \{P\}\, e\, \{Q\}$ guaranties that if the precondition P holds in m and $\mathbf{S}([\,], e, m) \to^* \mathbf{S}([\,], [v]_\Omega, m')$, then the postcondition $Q\,v$ holds in m'. Also, it ensures no undefined behavior occurs for each possible execution order. Soundness of the axiomatic semantics means that $\Delta \vdash \{P\}\, e\, \{Q\}$ implies $\Delta \vDash \{P\}\, e\, \{Q\}$ (and likewise for $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$).

We prove soundness (Theorem 6.8) by mutual induction on the derivations of $\Delta \vdash \{P\}\, e\, \{Q\}$ and $\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}$. Hence, $\Delta \vDash \{P\}\, e\, \{Q\}$ and $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$ should be

sufficiently strong so that we get appropriate induction hypotheses. The main difficulty is that the subexpressions e_l and e_r in $e_l \circledcirc e_r$ can do interleaved reduction steps. A simple minded definition of $\Delta \vDash \{P\}\, e\, \{Q\}$ that only talks about the end result of executing e does not work, because we also need to have information about states in between. Since the definitions of $\Delta \vDash \{P\}\, e\, \{Q\}$ and $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$ have a lot in common, we define a more general notion to factor out similarities.

Like Krebbers and Wiedijk [21], we have to enforce the reduction $\mathbf{S}(k,\ \phi,\ m) \twoheadrightarrow^* \mathbf{S}(k',\ \phi',\ m')$ to remain below a certain program context.

DEFINITION 6.1. *The* k-restricted reduction $S_1 \twoheadrightarrow_k S_2$ *is defined as* $S_1 \twoheadrightarrow S_2$ *provided* k *is a suffix of the program context of* S_2.

DEFINITION 6.2. *Given a predicate* $\bar{P} \subseteq \mathsf{stack} \times \mathsf{mem} \times \mathsf{focus}$, *the judgment* $\bar{P} \vDash_l^n \hat{\mathbf{S}}(k,\ \phi,\ m)$ *is inductively defined as:*

1. $\bar{P} \vDash_l^0 \hat{\mathbf{S}}(k,\ \phi,\ m)$
2. *If* $\bar{P}\,(\mathsf{getstack}\, l)\, m\, \phi$, *then* $\bar{P} \vDash_l^n \hat{\mathbf{S}}(l,\ \phi,\ m)$.
3. *If for all* m_f *with* $m \perp m_f$ *we have*
   (a) $\mathbf{S}(k,\ \phi,\ m \cup m_f)$ *is* $\twoheadrightarrow_l$*-reducible, and*
   (b) *if* $\mathbf{S}(k,\ \phi,\ m \cup m_f) \twoheadrightarrow_l S_2$, *then the state* S_2 *is of the shape* $\mathbf{S}(k_2,\ \phi_2,\ m_2 \cup m_f)$ *with* $m_2 \perp m_f$, $\phi \neq \overline{\mathsf{undef}}$, $\mathsf{locks}\ \phi_2 \subseteq \mathsf{locks}\ m_2$, *and* $\bar{P} \vDash_l^n \hat{\mathbf{S}}(k_2,\ \phi_2,\ m_2)$,
   then $\bar{P} \vDash_l^{1+n} \hat{\mathbf{S}}(k,\ \phi,\ m)$.

The intuitive meaning of $\bar{P} \vDash_l^n \hat{\mathbf{S}}(k,\ \phi,\ m)$ is that all $\twoheadrightarrow_l$-reductions paths of at most n steps starting at $\mathbf{S}(k,\ \phi,\ m \cup m_f)$:

- do not get stuck, and do not end up in the $\overline{\mathsf{undef}}$ state,
- always satisfy $\mathsf{locks}\ \phi \subseteq \mathsf{locks}\ m$ during the execution, and
- the end-state satisfies $\bar{P}$ and has program context l.

To handle interleaving of expressions, we allow the framing memory m_f to change at each step during the execution. Hence, instead of defining $\bar{P} \vDash_l^n \hat{\mathbf{S}}(k,\ \phi,\ m)$ using the reflexive transitive closure of $\twoheadrightarrow_l$, we defined it inductively using single steps. The condition $\mathsf{locks}\ \phi_2 \subseteq \mathsf{locks}\ m_2$ on the annotated locks with respect to the locks in the memory is used to separate the locks of subexpressions. We use a step-indexed approach to handle function calls.

The judgment $\bar{P} \vDash_l^n \hat{\mathbf{S}}(k,\ \phi,\ m)$ enjoys a nice composition property, and satisfies an abstract version of the weakening and frame rule of separation logic.

LEMMA 6.3. *Given contexts* k_1, k_2 *and* k_3 *with* k_2 *a suffix of* k_1 *and* k_3 *a suffix of* k_2. *If (a)* $\bar{P} \vDash_{k_2}^n \hat{\mathbf{S}}(k_1,\ \phi,\ m)$, *and (b) for all* m' *and* ϕ' *with* $\bar{P}\,(\mathsf{getstack}\, k_2)\, m'\, \phi'$ *we have* $\bar{Q} \vDash_{k_3}^n \hat{\mathbf{S}}(k_2,\ \phi',\ m')$, *then finally we have* $\bar{Q} \vDash_{k_3}^n \hat{\mathbf{S}}(k_1,\ \phi',\ m)$.

LEMMA 6.4. *Given memories* m *and* m_2 *such that* $m \perp m_2$. *If (a)* $\bar{P} \vDash_l^n \hat{\mathbf{S}}(k,\ \phi,\ m)$, *and (b) for all* m' *and* ϕ' *with* $m' \perp m_2$ *and* $\bar{P}\,(\mathsf{getstack}\, l)\, m'\ \phi'$ *we have* $\bar{Q}\,(\mathsf{getstack}\, l)\,(m' \cup m_2)\, \phi$, *then finally we have* $\bar{Q} \vDash_l^n \hat{\mathbf{S}}(k,\ \phi,\ m \cup m_2)$.

DEFINITION 6.5. Validity of the environment Δ, *notation* $\vDash_n \Delta$ *is defined as: for all* f *with* $\Delta\, f = (\forall \vec{y}\,.\, \forall \vec{v}\,.\, \{P\, \vec{y}\, \vec{v}\}\ \{Q\, \vec{y}\, \vec{v}\})$ *and* $P\, \vec{y}\, \vec{v}\, (\mathsf{getstack}\, k)\, m$ *we have* $\bar{Q}_{\vec{y},\vec{v}} \vDash_k^n \hat{\mathbf{S}}(\overline{\mathsf{call}}\, f\, \vec{v},\ k,\ m)$. *Here,* $\bar{Q}_{\vec{y},\vec{v}}$ *is defined as:*

$$\bar{Q}_{\vec{y},\vec{v}} := \lambda \rho\, \phi\, m'\,.\, \exists v\,.\, \phi = \overline{\mathsf{return}}\, v \wedge \mathsf{locks}\ m = \emptyset \wedge Q\, \vec{y}\, \vec{v}\, v\, \rho\, m'.$$

DEFINITION 6.6. Partial correctness of an expression e, *notation* $\Delta \vDash \{P\}\, e\, \{Q\}$ *is defined as: if* $\vDash_n \Delta$, $\mathsf{locks}\ e = \mathsf{locks}\ m = \emptyset$ *and* $P\, d\, (\mathsf{getstack}\, k)\, m$, *then* $\bar{Q} \vDash_k^n \hat{\mathbf{S}}(e,\ k,\ m)$. *Here* $\bar{Q}$ *is defined as:*

$$\bar{Q} := \lambda \rho\, \phi\, m'\,.\, \exists v\, \Omega\,.\, \phi = [v]_\Omega \wedge \mathsf{locks}\ m = \Omega \wedge Q\, v\, \rho\, m'.$$

Krebbers and Wiedijk [21] noticed that the assertions P, Q, J and R in $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$ correspond to the four directions $\searrow$, $\nearrow$, $\curvearrowright$ and $\Rsh$ in which traversal through a statement is performed. Hence, we treat $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$ as a triple $\Delta;\ \vec{P} \vDash s$, where $\vec{P}$ is a function from directions to assertions such that $\vec{P} \searrow = P$, $\vec{P} \nearrow = Q$, $\vec{P}\,(\curvearrowright l) = J\, l$ and $\vec{P}\,(\Rsh v) = R\, v$.

DEFINITION 6.7. Partial correctness of a statement s, *notation* $\Delta;\ \vec{P} \vDash s$ *is defined as: if* $\vDash_n \Delta$, $\mathsf{down}\ d\, s$, $\mathsf{locks}\ s = \mathsf{locks}\ m = \emptyset$ *and* $\vec{P}\, d\, (\mathsf{getstack}\, k)\, m$, *then* $\bar{P}_s \vDash_k^n \hat{\mathbf{S}}((d,\ s),\ k,\ m)$. *Here,* down *holds if* $\mathsf{down} \searrow s'$ *or* $\mathsf{down}\ (\curvearrowright l)\, s'$ *with* $l \in \mathsf{labels}\ s'$, *and* $\bar{P}_s$ *is defined as:*

$$\bar{P}_s := \lambda \rho\, \phi\, m'\,.\, \exists d'\, s'\,.\ \phi = (d',\ s') \wedge \neg \mathsf{down}\ d'\ s' \wedge \mathsf{locks}\ m = \emptyset \wedge \vec{P}\, d'\, \rho\, m'.$$

THEOREM 6.8 (Soundness). *We have:*

1. $\Delta \vdash \{P\}\, e\, \{Q\}$ *implies* $\Delta \vDash \{P\}\, e\, \{Q\}$, *and*
2. $\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}$ *implies* $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$.

This theorem is proven by mutual induction on the derivation of $\Delta \vdash \{P\}\, e\, \{Q\}$ and $\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}$. Thus, for each rule of the axiomatic semantics, we have to show that it holds in the model. In order to prove the (e-base) case, we need to show that the expression evaluation $[\![e]\!]_{\rho,m}$ is sound with respect to the operational semantics. To prove the cases of the other expressions constructs, we use the following generic lemma that deals with the subtleties of interleaving subexpressions.

LEMMA 6.9. *Given a singular expression context* $\mathcal{E}$ *with* u *holes and* $\mathsf{locks}\ E = \emptyset$, *memories* $\vec{m}$ *with* $\perp \vec{m}$, *expressions* $\vec{e}$ *with* $\mathsf{locks}\ e_i \subseteq \mathsf{locks}\ m_i$ *for each* $i < u$, *and functions of values to assertions* $\vec{P}$ *and* Q. *Now, if*

1. $\bar{P}_i \vDash_k^n \hat{\mathbf{S}}(k,\ e_i,\ m_i)$ *for each* $i < u$, *and*
2. *for all* $\vec{\Omega}$, $\vec{v}$ *and* $\vec{m}'$ *with* $\perp \vec{m}'$,
   (a) $\mathsf{locks}\ m_i = \Omega_i$ *for each* $i < u$, *and*
   (b) $P_i\, v_i\, k, m_i'$ *for each* $i < u$,
   we have $\bar{Q} \vDash_k^n \hat{\mathbf{S}}(k,\ E[\,[v_0]_{\Omega_0} \ldots [v_n]_{\Omega_n}\,],\ \bigcup \vec{m}')$,

then $\bar{Q} \vDash_k^n \hat{\mathbf{S}}(k,\ E[\,\vec{e}\,],\ \bigcup \vec{m})$.

The previous lemma is proven by induction on the number of steps n. Theorem 2.13 is used to reason about disjoint memories. The proofs of the cases for the statement judgments are quite similar to those by Krebbers and Wiedijk [21]; they involve chasing all possible reduction paths and use Lemma 6.3. We refer to the Coq formalization for the actual proofs.

7. Extensions

In this section we describe two extensions of our axiomatic semantics that improve handling of function calls. The first extension makes it easier to deal with pure functions (*i.e.* functions that have no side-effects), whereas the second extension enables different function calls in the same expression to have access to a shared writable part of the memory. These extensions do not change the memory model or the operational semantics.

We make it possible to associate a mathematical function to a pure function. For example, this can be used to smoothly use Coq's `gcd` function to reason about its counterpart in C. We extend the environment Δ to map function names to:

1. Specifications of pure functions by partial Coq functions, or,
2. Specifications of impure functions using their pre- and postcondition $\forall \vec{y}\, \forall \vec{v}\,.\, \{P\, \vec{y}\, \vec{v}\}\ \{Q\, \vec{y}\, \vec{v}\}$.

The (add funs) rule is extended so that a pure function f with corresponding Coq function $F : \mathsf{list\ val} \to \mathsf{val^{opt}}$ can be added to Δ by proving $\forall \vec{v} \,.\, \{\mathsf{emp} \wedge F\,\vec{v} \neq \bot\}\,\{\lambda v \,.\, \mathsf{emp} \wedge F\,\vec{v} = v\}$. Assertions are made parametric with respect to the environment Δ so that pure functions can be given an interpretation by $[\![e]\!]_{\rho,m}$. The (e-base) rule becomes:

$$\frac{P \vDash_\Delta e \Downarrow v}{\Delta \vdash \{P\}\, e\, \{\lambda v' \,.\, v = v' \wedge P\}}$$

Apart from the parametrization by Δ, most rules of the axiomatic semantics remain unchanged. To prove soundness of the (e-base) rule, we now also have to deal with pure function calls.

The rules for expressions of our axiomatic semantics require the memory to be separated into disjoint parts for the subexpressions e_l and e_r at each operator $e_l \circledcirc e_r$. Hence, only read-only memory can be shared by function calls that appear in both e_l and e_r. This is not very satisfactory, as functions (even when used in the same expression) often need to have access to shared data structures (*e.g.* a buffer or hash-table). To that end, we extend the expression judgment $\Delta;\ B \vdash \{P\}\, e\, \{Q\}$ with an assertion B that can be used to describe the invariant of shared memory by all function calls in e. The following frame rule can be used to move the memory out of the pre- and postcondition into the assertion B.

$$\frac{\Delta;\ A * B \vdash \{P\}\, e\, \{Q\}}{\Delta;\ B \vdash \{P * A\}\, e\, \{Q * A\}}$$

The rule (e-call) for function calls is changed so that B can be used to prove the precondition, and so that B has to be reobtained from the postcondition when the function call is finished.

For the soundness proof, we need to generalize Definition 6.2 so that the memory of B is part of the framing memory m_f during execution of the expression, and so that is will be transferred to the active memory at a function call. This extension still does not give completeness of the axiomatic semantics though. Consider:

```
int x = 0;
int f(int y) { return (x = y); }
int main() { f(3) + f(4); return x; }
```

Since the invariant B should hold before, after, and in between the function calls in the expression `f(3) + f(4)`, the best choice for it is $\mathtt{x} \overset{\mathsf{W}}{\mapsto} 0 \vee \mathtt{x} \overset{\mathsf{W}}{\mapsto} 3 \vee \mathtt{x} \overset{\mathsf{W}}{\mapsto} 4$. Hence, one can only prove that the program returns 0, 3 or 4 in the end, whereas it actually returns 3 or 4. We believe programs as the above are artificial, and such non-determinism is not frequent in actual C programs.

8. Formalization in Coq

All proofs in this paper have been fully formalized using the Coq proof assistant. Formalization has been of great help in order to develop and debug the semantics. We used Coq's notation mechanism combined with unicode symbols and type classes for overloading to let the Coq development correspond as well as possible to the definitions in this paper. However, in this paper, we presented the axiomatic semantics as an inference system, and showed that it has a model. Since we did not consider completeness, we directly proved all rules to be derivable with respect to the model.

We used Coq's type classes to provide abstract interfaces for commonly used structures like finite sets and finite partial functions, so that we were able to prove theory and implement automation in an abstract way. Our approach is greatly inspired by the *unbundled* approach of Spitters and van der Weegen [33]. However, whereas their work heavily relies on *setoids* (types equipped with an equivalence relation), we tried to avoid that by using Leibniz equality as much as possible.

In particular, our interface for finite partial functions requires *extensionality* with respect to Leibniz equality, *i.e.* $m_1 = m_2$ iff $\forall x \,.\, m_1\, x = m_2\, x$. Extensional equality of finite partial functions is particularly useful for dealing with assertions, which are defined as predicates on the stack and memory (Definition 5.2). Due to extensionality, we did not have to equip assertions with a proof that they respect extensional equality on memories.

Although intensional type theories like Coq do not satisfy extensionality, finite functions indexed by a countable type can still be implemented in a way that extensionality holds. This is achieved by representing finite functions as trees in canonical form.

Coq's support for dependent types has been particularly useful to formulate Lemma 6.9 where we have to deal with expression contexts with multiple holes. We represented these expression contexts using a type indexed by the number of holes.

Because the semantics described in this paper is rather big, it is quite cumbersome to prove properties about it without automation. In particular, the reduction $\rightarrowtriangle$ (Definition 4.10) is defined as an inductive type consisting of 33 constructors. To this end, we have automated many steps of the proofs. For example, we implemented a tactic `do_cstep` to automatically perform reduction steps and to solve the required side-conditions, a tactic `inv_cstep` to perform case analyzes on reductions and to automatically discharge impossible cases, and a tactic `solve_mem_disjoint` to automatically prove disjointness of memories using the algebraic method described in Section 2. Ongoing experiments show that this approach is successful, as the semantics can be extended easily without having to redo many proofs.

Our Coq code, available at `http://robbertkrebbers.nl/research/ch2o`, is about 10 000 lines of code including comments and white space. Apart from that, our library on general purpose theory (finite sets, finite functions, lists, *etc.*) is about 9 000 lines.

9. Conclusions and further research

The further reaching goal of this work is to develop an operational and axiomatic semantics for a large part of the C11 programming language [20]. Formal treatment of non-determinism and sequence points in expressions with side-effects is a necessary step towards this goal. Our next step is to integrate the work of Krebbers [19] on the C memory model and type system into our formalization, and make the language typed. Once integrated, we intend to develop a verified type checker and interpreter so we can test the semantics using actual C programs.

Extending our operational semantics to deal with concurrency is also an interesting topic for future research. For example, it would be useful to investigate whether our semantics can be extended to weak memories in the same way as CompCert has been extended to CompCert TSO [32].

In order to turn the theory presented in this paper into an actual tool for verification of C programs, we need to develop a verification condition generator. A verification condition generator takes a program with logical annotations and generates a set of verification conditions that need to be verified. For ordinary Hoare logic, the most common approach is to use a variant of Dijkstra's weakest precondition calculus. For separation logic, one typically uses symbolic execution (see for example Berdine *et al.* [4]). In case of our axiomatic semantics, it is not directly clear whether any of these two approaches can be applied. The problematic part is that in the rules for binary operations, one has to split the memory in two parts using the separating conjunction $*$. It would be interesting to investigate how this can be automated.

Another challenge to use our axiomatic semantics for program verification is strong automation for separation logic. Specific to the Coq proof assistant there has been work on this by for example Appel [1], Chlipala [10], and Bengtson *et al.* [3].

As shown in Section 7, our axiomatic semantics is not complete with respect to the operational semantics. Nonetheless, as demon-

strated in Section 5, we can derive the ordinary rules of separation logic, and rules specifically tailored for certain constructs. It would be useful to investigate whether it is complete for another variant of the operational semantics.

Due to our *local* treatment of locks to model the sequence point restriction, our operational semantics assigns undefined behavior to more programs than the C11 standard does (see page 5). We believe that this is a reasonable trade-off, because it brings the operational semantics closer to the axiomatic semantics, and only makes artificial programs illegal. Moreover, this treatment of locks may enable some useful optimizations that are not allowed by the C11 standard.

A particular optimization that is not justified by the C11 standard is CompCert [23]'s passing by reference of struct and union values through expressions. In CompCert, copies of struct and union values are made only at function calls and assignments. Let us take a look at the following example.

```
struct S { int x; } s1 = { 1 }, s2 = { 2 };
int f() {
  if (s1.x == 2) { s2.x = 40; } return 0;
}
int main() { return (s1 = s2).x + f(); }
```

The execution order where the CompCert semantics deviates from the C11 standard is: (a) perform the assignment `s1 = s2` whose result is a reference to `s2` instead of a copy of `s2`, (b) call `f`, which uses an `if (s1.x == 2)` to detect that the assignment `s1 = s2` has been executed, and finally (c) the field `x` of the struct that has been modified by `f` is taken. The return value for this execution order is 40, which cannot be obtained from any execution order if structs were passed by value.

Using a global treatment of sequence points (as in Ellison and Rosu [12] or Norrish [26]), no execution order of this program leads to a sequence point violation. However, with our local treatment of sequence points, the execution order described above will exhibit a sequence point violation, and hence our treatment would justify by reference passing of struct values for this program. It would be interesting to investigate whether our semantics of sequence points justifies by reference passing of struct and union values through expressions for arbitrary programs.

Acknowledgments

I thank my advisors Freek Wiedijk and Herman Geuvers, and the anonymous referees for their helpful suggestions. I am indebted to Xavier Leroy for many useful discussions. This work is financed by the Netherlands Organisation for Scientific Research (NWO).

References

[1] A. W. Appel. Tactics for Separation Logic, 2006. Available at `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`.

[2] A. W. Appel and S. Blazy. Separation Logic for Small-Step Cminor. In *TPHOLs*, volume 4732 of *LNCS*, pages 5–21, 2007.

[3] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! - A Framework for Higher-Order Separation Logic in Coq. In *ITP*, volume 7406 of *LNCS*, pages 315–331, 2012.

[4] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68, 2005.

[5] P. E. Black and P. J. Windley. Inference Rules for Programming Languages with Side Effects in Expressions. In *TPHOLs*, volume 1125 of *LNCS*, pages 51–60, 1996.

[6] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, 2005.

[7] J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72, 2003.

[8] C. Calcagno, P. W. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic. In *LICS*, pages 366–378, 2007.

[9] B. Campbell. An Executable Semantics for CompCert C. In *CPP*, volume 7679 of *LNCS*, pages 60–75, 2012.

[10] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011.

[11] 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.

[12] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[13] C. Ellison and G. Rosu. Slides of [12], 2012. `http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl-slides.pdf`.

[14] M. Felleisen, D. P. Friedman, E. E. Kohlbecker, and B. F. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[15] P. Herms. *Certification of a Tool Chain for Deductive Program Verification*. PhD thesis, l'Université Paris-Sud, 2013.

[16] International Organization for Standardization. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.

[17] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[18] R. Krebbers. Non-determinism and sequence points in C (blog post), 2013. Available at `http://gallium.inria.fr/blog/non-determinism-and-sequence-points-in-c/`.

[19] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*, 2013.

[20] 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.

[21] 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.

[22] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[23] X. Leroy. The CompCert verified compiler, software and commented proof. Available at `http://compcert.inria.fr/`, 2012.

[24] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research report RR-7987, INRIA, 2012.

[25] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[26] M. Norrish. Deterministic Expressions in C. In *ESOP*, volume 1576 of *LNCS*, pages 147–161, 1999.

[27] P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, volume 3170 of *LNCS*, pages 49–67, 2004.

[28] 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.

[29] D. v. Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[30] N. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.

[31] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.

[32] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM*, 60(3):22, 2013.

[33] B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4): 795–825, 2011.