Separation Logic for Non-local Control Flow and Block Scope Variables

Robbert Krebbers and Freek Wiedijk

ICIS, Radboud University Nijmegen, The Netherlands

Abstract. We present an approach for handling non-local control flow (goto and return statements) in the presence of allocation and deallocation of block scope variables in imperative programming languages.
We define a small step operational semantics and an axiomatic semantics (in the form of a separation logic) for a small C-like language that combines these two features, and which also supports pointers to block scope variables. Our operational semantics represents the program state through a generalization of Huet's zipper data structure.
We prove soundness of our axiomatic semantics with respect to our operational semantics. This proof has been fully formalized in Coq.

1 Introduction

There is a gap between programming language features that can be described well by a formal semantics, and those that are available in widely used programming languages. This gap needs to be bridged in order for formal methods to become mainstream. However, interaction between the more 'dirty' features of widely used programming languages tends to make an accurate formal semantics even more difficult. An example of such interaction is the goto statement in the presence of block scope variables in the C programming language.

C allows unrestricted gotos which (unlike break and continue) may not only jump out of blocks, but can also jump into blocks. Orthogonal to this, blocks may contain local variables, which can be "taken out of their block" by keeping a pointer to them. This makes non-local control in C (including break and continue) even more unrestricted, as leaving a block results in the memory of these variables being freed, and thus making pointers to them invalid. Consider:

```
int *p = NULL;
l: if (p) { return (*p); }
   else { int j = 10; p = &j; goto l; }
```

Here, when the label `l` is passed for the first time, the variable `p` is `NULL`. Hence, execution continues in the block containing `j` where `p` is assigned a pointer to `j`. However, after the `goto l` statement, the block containing `j` is left, and the memory of `j` is freed. After this, dereferencing `p` is no longer legal, making the program exhibit *undefined behavior*.

If a program exhibits undefined behavior, the ISO C standard [8] allows it to do literally anything. This is to avoid compilers having to insert (possibly

expensive) dynamic checks to handle corner cases. In particular, in the case of non-local control flow this means that an implementation can ignore allocation issues when jumping, but a semantics cannot. Not describing certain undefined behaviors would therefore mean that some programs can be proven correct with respect to the formal semantic whereas they may crash or behave unexpectedly when compiled with an actual C compiler.

It is well known that a small step semantics is more flexible than a big step semantics for modeling more intricate programming language features. In a small step semantics, it is nonetheless intuitive to treat uses of goto as big steps, as executing them makes the program jump to a totally different place in one step. For functional languages, there has been a lot of research on modeling control (`call/cc` and variants thereof) in a purely small step manner (see [6] for example). This indicates that the intuition that uses of non-local control should be treated as big steps is not correct.

We show that a purely small step semantics is also better suited to handle the interaction between gotos and block scope variables in imperative programming languages. Our semantics lets the goto statement traverse in small steps through the program to search for its corresponding label. The required allocations and deallocations are calculated incrementally during this traversal.

Our choice of considering goto at all may seem surprising. Since the revolution of structured programming in the seventies, many people have considered goto as bad programming practice [4]. However, some have disagreed [9], and gotos are still widely used in practice. For example, the current Linux kernel contains about a hundred thousand uses of goto. Goto statements are particularly useful for breaking from multiple nested loops, and for systematically cleaning up resources after an error occurred. Also, gotos can be used to increase performance.

Approach. We define a small step operational semantics for a small C-like language that supports both non-local control flow and block scope variables. To obtain more confidence in this semantics, and to support reasoning about programs in this language, we define an axiomatic semantics for the same fragment, and prove its soundness with respect to the operational semantics.

Our operational semantics uses a *zipper*-like data structure [7] to store both the location of the substatement that is being executed and the program stack. Because we allow pointers to local variables, the stack contains references to the value of each variable instead of the value itself. Execution of the program occurs by traversal through the zipper in one of the following directions: *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 zipper until the label l has been reached.

Related work. Goto statements (and other forms of non-local control) are often modeled using continuations. Appel and Blazy provide a small step continuation semantics for Cminor [2] that supports return statements. CompCert extends their approach to support goto statements in Cmedium [12]. Ellison and Rosu [5] also use continuations to model gotos in their C11 semantics, but whereas the

CompCert semantics does not support block scope variables, they do. We further discuss the differences between continuations and our approach in Section 3.

Tews [18] defines a denotational semantics for a C-like language that supports goto and unstructured switch statements. His state includes variants for non-local control corresponding to our directions $\curvearrowright$ and $\twoheadrightarrow$.

The most closely related work to our axiomatic semantics is Appel and Blazy's separation logic for Cminor in Coq [2]. Their separation logic supports return statements, but does not support gotos, nor block scope variables. Von Oheimb [15] defines an operational and axiomatic semantics for a Java-like language in Isabelle. His language supports both local variables and mutually recursive function calls. Although his work is fairly different from ours, our approach to mutual recursion is heavily inspired by his. Furthermore, Chlipala [3] gives a separation logic for a low-level language in Coq that supports gotos. His approach to automation is impressive, but he does not give an explicit operational semantics and does not consider block scope variables.

Contribution. Our contribution is threefold:

- We define a small step operational semantics using a novel zipper based data structure to handle the interaction between gotos and block scope variables in a correct way (Section 2 and 3).
- We give an axiomatic semantics that allows reasoning about programs with gotos, pointers to local variables, and mutually recursive function calls. We demonstrate it by verifying Euclid's algorithm (Section 4).
- We prove the soundness of our axiomatic semantics (Section 5). This proof has been fully formalized in the Coq proof assistant (Section 6).

2 The language

Our memory is a finite partial function from natural numbers to values, where a value is either an unbounded integer, a pointer represented by a natural number corresponding to the index of a memory cell, or the `NULL`-pointer.

Definition 2.1. *A* partial function *from A to B is a (total) function from A to B^{opt}, where A^{opt} is the* option type*, defined as containing either $\bot$ or x for some $x \in A$. 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.* Disjointness*, notation $f_1 \perp f_2$, is defined as $\forall x\,.\, f_1\, x = \bot \vee f_2\, x = \bot$. Given f_1 and f_2 with $f_1 \perp f_2$, the operation $f_1 \cup f_2$ yields their* union*. Moreover, the* inclusion *$f_1 \subseteq f_2$ is defined as $\forall x\, y\,.\, f_1\, x = y \to f_2\, x = y$.*

Definition 2.2. Values *are defined as:*

$$v ::= \texttt{int}\ n \mid \texttt{ptr}\ b \mid \texttt{NULL}$$

Memories *(typically named m) are finite partial functions from natural numbers to values. A value v is* true*, notation* istrue *v, if it is of the shape* `int` *n with $n \neq 0$, or* `ptr` *b. It is* false*, notation* isfalse *v, otherwise.*

Expressions are side-effect free and will be given a deterministic semantics by an evaluation function. The variables used in expressions are De Bruijn indexes, *i.e.* the variable x_i refers to the ith value on the stack. De Bruijn indexes 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.

Definition 2.3. Expressions *are defined as:*

$$\begin{aligned} \circledcirc &::= == \mid \leq \mid + \mid * \mid / \mid \% \\ e &::= x_i \mid v \mid \texttt{load}\, e \mid e_1 \circledcirc e_2 \end{aligned}$$

Stacks (typically named ρ) are lists of memory indexes rather than lists of values. This allows us to treat pointers to both local and allocated storage in a uniform way. Evaluation of a variable thus consists of looking up its address in the stack, and returning a pointer to that address.

Definition 2.4. Evaluation $[\![e]\!]_{\rho,m}$ of an expression e *in a stack* ρ *and memory* m *is defined by the following partial function:*

$$\begin{aligned} [\![x_i]\!]_{\rho,m} &:= \texttt{ptr}\, a \qquad \textit{if } \rho\, i = a \\ [\![v]\!]_{\rho,m} &:= v \\ [\![\texttt{load}\, e]\!]_{\rho,m} &:= m\, a \qquad \textit{if } [\![e]\!]_{\rho,m} = \texttt{ptr}\, a \\ [\![e_1 \circledcirc e_2]\!]_{\rho,m} &:= [\![e_1]\!]_{\rho,m} \circledcirc [\![e_2]\!]_{\rho,m} \end{aligned}$$

Lemma 2.5. *If* $m_1 \subseteq m_2$ *and* $[\![e]\!]_{\rho,m_1} = v$*, then* $[\![e]\!]_{\rho,m_2} = v$*.*

Definition 2.6. Statements *are defined as:*

$$\begin{aligned} s ::= {} & \texttt{block}\, s \mid e_l := e_r \mid f(\vec{e}) \mid \texttt{skip} \mid \texttt{goto}\, l \\ & \mid l : s \mid s_1 \,;\, s_2 \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{return} \end{aligned}$$

The construct `block` s opens a new scope with one variable. Since we use De Bruijn indexes for variables, it does not contain the name of the variable. For presentation's sake, we have omitted functions that return values (these are however included in our Coq formalization). In the semantics presented here, an additional function parameter with a pointer for the return value can be used instead. Given a statement s, the function labels s collects the labels of labeled statements in s, and the function gotos s collects the labels of gotos in s.

3 Operational semantics

We define the semantics of 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*. This reduction relation traverses through

the program in small steps by moving the focus on the substatement that is being executed. Uses of non-local control (goto and return) are performed in small steps rather than in big steps as well.

In order to model the concept of focusing on the substatement that is being executed, we need a data structure to capture the location in the program. For this we define *program contexts* as an extension of Huet's zipper data structure [7]. Program contexts extend the zipper data structure 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 [2,11,12]). 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 (see the Coq formalization).

The fact that program contexts do not throw away the parts of the statement that have been executed is essential for our 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 went 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.

In a continuation semantics, upon the use of a goto, one typically computes, or looks up, the statement and continuation corresponding to the target label. However, it is not very natural to reconstruct the required allocations and deallocations from the current and target continuations.

Definition 3.1. Singular statement contexts *are defined as:*

$$E_{\mathcal{S}} ::= \Box\,;s_2 \mid s_1\,;\Box \mid \texttt{if}\ (e)\ \Box\ \texttt{else}\ s_2 \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ \Box \mid l:\Box$$

Given a singular statement context $E_{\mathcal{S}}$ and a statement s, substitution of s for the hole in $E_{\mathcal{S}}$*, notation $E_{\mathcal{S}}[\,s\,]$, is defined in the ordinary way.*

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

Definition 3.2. Singular program contexts *are defined as:*

$$E ::= E_{\mathcal{S}} \mid \texttt{block}_b\ \Box \mid \mathsf{call}\ f\ \vec{e} \mid \mathsf{params}\ \vec{b}$$

Program contexts *(typically named k) are lists of singular program contexts.*

The previously introduced contexts will be used as follows.

- When entering a block, $\texttt{block}\ s$, the context $\texttt{block}_b\ \square$ is appended to the head of the program context. It associates the block scope variable with its corresponding memory index b.
- Upon a function call, $f(\vec{e})$, the context $\mathsf{call}\ f\ \vec{e}$ is appended to the head of the program context. It contains the location of the caller so that it can be restored when the called function f returns.
- When a function body is entered, the context $\mathsf{params}\ \vec{b}$ is appended to the head of the program context. It contains a list of memory indexes of the function parameters.

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

Definition 3.3. *The* corresponding stack $\mathsf{getstack}\ k$ *of* k *is defined as:*

$$\begin{aligned}
\mathsf{getstack}\ (E_{\mathcal{S}} :: k) &:= \mathsf{getstack}\ k\\
\mathsf{getstack}\ (\texttt{block}_b\ \square :: k) &:= b :: \mathsf{getstack}\ k\\
\mathsf{getstack}\ (\mathsf{call}\ f\ \vec{e} :: k) &:= [\,]\\
\mathsf{getstack}\ (\mathsf{params}\ \vec{b} :: k) &:= \vec{b} +\!\!+ \mathsf{getstack}\ k
\end{aligned}$$

We will treat getstack *as an implicit coercion and will omit it everywhere.*

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

Definition 3.4. Directions, focuses *and* program states *are defined as:*

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

A program state $\mathbf{S}(k,\ \phi,\ m)$ consists of a program context k, the part of the program that is focused ϕ, and the memory m. Like Leroy's semantics for Cminor [11], we consider three kinds of states: (a) execution of statements (b) calling a function (c) returning from a function. The CompCert Cmedium semantics [12] also includes a state for execution of expressions and a *stuck* state for undefined behavior. Since our expressions are side-effect free, we do not need an additional expression state. Furthermore, since expressions are deterministic, we can easily capture undefined behavior by letting the reduction get stuck.

Definition 3.5. *The relation* $\texttt{allocparams}\ m_1\ \vec{b}\ \vec{v}\ m_2$ *(non-deterministically)* allocates fresh blocks $\vec{b}$ for function parameters $\vec{v}$. *It is inductively defined as:*

$$\frac{}{\texttt{allocparams}\ m\ [\,]\ [\,]\ m} \qquad \frac{\texttt{allocparams}\ m_1\ \vec{b}\ \vec{v}\ m_2 \qquad m_2\, b = \bot}{\texttt{allocparams}\ m_1\ (b :: \vec{b})\ (v :: \vec{v})\ m_2[b := v]}$$

Definition 3.6. *Given a function δ assigning statements to function names, the* small step reduction relation $S_1 \rightarrowtriangle S_2$ *is defined as:*

1. For simple statements:
   (a) $\mathbf{S}(k, (\searrow, e_1 := e_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, e_1 := e_2), m[a := v])$
   for any a and v such that $[\![e_1]\!]_{k,m} = \mathtt{ptr}\ a$, $[\![e_2]\!]_{k,m} = v$ and $m\, a \neq \bot$.
   (b) $\mathbf{S}(k, (\searrow, f(\vec{e})), m) \rightarrowtriangle \mathbf{S}(\mathsf{call}\ f\ \vec{e} :: k, \overline{\mathsf{call}}\ f\ \vec{v}, m)$
   for any $\vec{v}$ such that $[\![e_i]\!]_{k,m} = v_i$ for each i.
   (c) $\mathbf{S}(k, (\searrow, \mathtt{skip}), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, \mathtt{skip}), m)$
   (d) $\mathbf{S}(k, (\searrow, \mathtt{goto}\ l), m) \rightarrowtriangle \mathbf{S}(k, (\curvearrowright l, \mathtt{goto}\ l), m)$
   (e) $\mathbf{S}(k, (\searrow, \mathtt{return}), m) \rightarrowtriangle \mathbf{S}(k, (\Uparrow, \mathtt{return}), m)$
2. For compound statements:
   (a) $\mathbf{S}(k, (\searrow, \mathtt{block}\ s), m) \rightarrowtriangle \mathbf{S}((\mathtt{block}_b\ \square) :: k, (\searrow, s), m[b := v])$
   for any b and v such that $m\, b = \bot$.
   (b) $\mathbf{S}(k, (\searrow, s_1\,; s_2), m) \rightarrowtriangle \mathbf{S}((\square\,; s_2) :: k, (\searrow, s_1), m)$
   (c) $\mathbf{S}(k, (\searrow, \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2), m) \rightarrowtriangle \mathbf{S}((\mathtt{if}\ (e)\ \square\ \mathtt{else}\ s_2) :: k, (\searrow, s_1), m)$
   for any v such that $[\![e]\!]_{k,m} = v$ and istrue *v.*
   (d) $\mathbf{S}(k, (\searrow, \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2), m) \rightarrowtriangle \mathbf{S}((\mathtt{if}\ (e)\ s_1\ \mathtt{else}\ \square) :: k, (\searrow, s_2), m)$
   for any v such that $[\![e]\!]_{k,m} = v$ and isfalse *v.*
   (e) $\mathbf{S}(k, (\searrow, l : s), m) \rightarrowtriangle \mathbf{S}((l : \square) :: k, (\searrow, s), m)$
   (f) $\mathbf{S}((\mathtt{block}_b\ \square) :: k, (\nearrow, s), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, \mathtt{block}\ s), m[b := \bot])$
   (g) $\mathbf{S}((\square\,; s_2) :: k, (\nearrow, s_1), m) \rightarrowtriangle \mathbf{S}((s_1\,; \square) :: k, (\searrow, s_2), m)$
   (h) $\mathbf{S}((s_1\,; \square) :: k, (\nearrow, s_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, s_1\,; s_2), m)$
   (i) $\mathbf{S}((\mathtt{if}\ (e)\ \square\ \mathtt{else}\ s_2) :: k, (\nearrow, s_1), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2), m)$
   (j) $\mathbf{S}((\mathtt{if}\ (e)\ s_1\ \mathtt{else}\ \square) :: k, (\nearrow, s_2), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2), m)$
   (k) $\mathbf{S}((l : \square) :: k, (\nearrow, s), m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, l : s), m)$
3. For function calls:
   (a) $\mathbf{S}(k, \overline{\mathsf{call}}\ f\ \vec{v}, m_1) \rightarrowtriangle \mathbf{S}(\mathsf{params}\ \vec{b} :: k, (\searrow, s), m_2)$
   for any s, $\vec{b}$ and m_2 such that $\delta\, f = s$ and `allocparams` *$m_1\ \vec{b}\ \vec{v}\ m_2$.*
   (b) $\mathbf{S}(\mathsf{params}\ \vec{b} :: k, (\nearrow, s), m) \rightarrowtriangle \mathbf{S}(k, \overline{\mathsf{return}}, m[\vec{b} := \vec{\bot}])$
   (c) $\mathbf{S}(\mathsf{params}\ \vec{b} :: k, (\Uparrow, s), m) \rightarrowtriangle \mathbf{S}(k, \overline{\mathsf{return}}, m[\vec{b} := \vec{\bot}])$
   (d) $\mathbf{S}(\mathsf{call}\ f\ \vec{e} :: k, \overline{\mathsf{return}}, m) \rightarrowtriangle \mathbf{S}(k, (\nearrow, f(\vec{e})), m)$
4. For non-local control flow:
   (a) $\mathbf{S}((\mathtt{block}_b\ \square) :: k, (\Uparrow, s), m) \rightarrowtriangle \mathbf{S}(k, (\Uparrow, \mathtt{block}\ s), m[b := \bot])$
   (b) $\mathbf{S}(E_{\mathcal{S}} :: k, (\Uparrow, s), m) \rightarrowtriangle \mathbf{S}(k, (\Uparrow, E_{\mathcal{S}}[\, s\,]), m)$
   (c) $\mathbf{S}(k, (\curvearrowright l, \mathtt{block}\ s), m) \rightarrowtriangle \mathbf{S}((\mathtt{block}_b\ \square) :: k, (\curvearrowright l, s), m[b := v])$
   for any b and v such that $m\, b = \bot$, and provided that $l \in$ labels *s.*
   (d) $\mathbf{S}(k, (\curvearrowright l, l : s), m) \rightarrowtriangle \mathbf{S}((l : \square) :: k, (\searrow, s), m)$
   (e) $\mathbf{S}(k, (\curvearrowright l, E_{\mathcal{S}}[\, s\,]), m) \rightarrowtriangle \mathbf{S}(E_{\mathcal{S}} :: k, (\curvearrowright l, s), m)$ *provided that $l \in$* labels *s.*
   (f) $\mathbf{S}(\mathtt{block}_b\ \square :: k, (\curvearrowright l, s), m) \rightarrowtriangle \mathbf{S}(k, (\curvearrowright l, \mathtt{block}\ s), m[b := \bot])$
   provided that $l \notin$ labels *s.*
   (g) $\mathbf{S}(E_{\mathcal{S}} :: k, (\curvearrowright l, s), m) \rightarrowtriangle \mathbf{S}(k, (\curvearrowright l, E_{\mathcal{S}}[\, s\,]), m)$ *provided that $l \notin$* labels *s.*

Note that the rules 4d and 4e overlap, and that the splitting into $E_{\mathcal{S}}$ and s in rule 4e is non-deterministic. We let $\rightarrow\!\!\triangleright^*$ denote the reflexive-transitive closure, and $\rightarrow\!\!\triangleright^n$ paths of $\leq n$ 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 to be executed. Consider the example from the introduction (with the return expression omitted).

```
int *p = NULL;
l: if (p) { return; }
   else { int j = 10; p = &j; goto l; }
```

Figure 1 below displays some states corresponding to execution of this program starting at `p = &j` in downwards direction.

Execution of a function call $\mathbf{S}(k,\,(\searrow,\,f(\vec{e})),\,m)$ consists of two reductions. The reduction to $\mathbf{S}(\mathsf{call}\ f\ \vec{e} :: k,\,\overline{\mathsf{call}}\ f\,\vec{v},\,m)$ evaluates the function parameters $\vec{e}$ to values $\vec{v}$, and stores the location of the calling function on the program context. The subsequent reduction to $\mathbf{S}(\mathsf{params}\ \vec{b} :: \mathsf{call}\ f\ \vec{e} :: k,\,(\searrow,\,s),\,m')$ looks up the called function's body s, allocates storage for the parameters $\vec{v}$, and then performs a transition to execute the called function's body.

We consider two directions for non-local control flow: *jump* $\curvearrowright l$ and *top* $\Uparrow$. After a `goto` l the direction $\curvearrowright l$ is used to traverse to the substatement labeled l. Although this search is non-deterministic, there are some side conditions on it so as to ensure it not going back and forth between the same locations. This is required as it otherwise may impose non-terminating behavior on terminating programs. The non-determinism could be removed entirely by adding additional side conditions. However we omitted doing so in order to ease formalization.

The direction $\Uparrow$ is used to traverse to the *top* of the statement after a `return`. When it reaches the top, there will be two reductions to leave the called function. The first reduction, from $\mathbf{S}(\mathsf{params}\ \vec{b} :: \mathsf{call}\ f\ \vec{e} :: k,\,(\Uparrow,\,s),\,m)$ to $\mathbf{S}(\mathsf{call}\ f\ \vec{e} :: k,\,\overline{\mathsf{return}},\,m[\vec{b} := \vec{\bot}])$, deallocates the function parameters, and the second, to $\mathbf{S}(k,\,(\nearrow,\,f(\vec{e})),\,m)$, reinstates the calling function.

$k_1 = \square\,;\texttt{goto}\ l$
$\quad :: x_0 := \texttt{int}\ 10\,;\square$
$\quad :: \texttt{block}_{b_j}\ \square$
$\quad :: \texttt{if}\ (\texttt{load}\ x_0)\ \texttt{return}$
$\quad\ \ \texttt{else}\ \square$
$\quad :: l : \square$
$\quad :: x_0 := \texttt{NULL}\,;\square$
$\quad :: \texttt{block}_{b_p}\ \square$
$\phi_1 = (\searrow,\, x_1 := x_0)$
$m_1 = \{b_p \mapsto \texttt{NULL}, b_j \mapsto 10\}$
$S_1 = \mathbf{S}(k_1,\,\phi_1,\,m_1)$

$k_2 = \square\,;\texttt{goto}\ l$
$\quad :: x_0 := \texttt{int}\ 10\,;\square$
$\quad :: \texttt{block}_{b_j}\ \square$
$\quad :: \texttt{if}\ (\texttt{load}\ x_0)\ \texttt{return}$
$\quad\ \ \texttt{else}\ \square$
$\quad :: l : \square$
$\quad :: x_0 := \texttt{NULL}\,;\square$
$\quad :: \texttt{block}_{b_p}\ \square$
$\phi_2 = (\nearrow,\, x_1 := x_0)$
$m_2 = \{b_p \mapsto \texttt{ptr}\ b_j, b_j \mapsto 10\}$
$S_2 = \mathbf{S}(k_2,\,\phi_2,\,m_2)$

$k_3 = x_1 := x_0\,;\square$
$\quad :: x_0 := \texttt{int}\ 10\,;\square$
$\quad :: \texttt{block}_{b_j}\ \square$
$\quad :: \texttt{if}\ (\texttt{load}\ x_0)\ \texttt{return}$
$\quad\ \ \texttt{else}\ \square$
$\quad :: l : \square$
$\quad :: x_0 := \texttt{NULL}\,;\square$
$\quad :: \texttt{block}_{b_p}\ \square$
$\phi_3 = (\searrow,\, \texttt{goto}\ l)$
$m_3 = \{b_p \mapsto \texttt{ptr}\ b_j, b_j \mapsto 10\}$
$S_3 = \mathbf{S}(k_3,\,\phi_3,\,m_3)$

Fig. 1. An example reduction path $S_1 \rightarrow\!\!\triangleright\ S_2 \rightarrow\!\!\triangleright\ S_3$.

When we relate our operational and axiomatic semantics in Section 5, we will have to restrict the traversal through the program to remain below a certain context.

Definition 3.7. *The* k-restricted reduction $S_1 \rightarrowtriangle_k S_2$ *is defined as* $S_1 \rightarrowtriangle S_2$ *provided that* k *is a suffix of the program context of* S_2*. We let* $\rightarrowtriangle_k^*$ *denote the reflexive-transitive closure, and* $\rightarrowtriangle_k^n$ *paths of* $\leq n$ *steps.*

Lemma 3.8. *If* $\mathbf{S}(k, (d, s), m) \rightarrowtriangle_k^* \mathbf{S}(k, (d', s'), m')$*, then* $s = s'$*.*

The previous lemma shows that the small step semantics indeed behaves as traversing through a zipper. Its proof is not entirely trivial due to the presence of function calls, as these add the statement of the called function to the state.

4 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 state before executing s, and execution of s terminates, then Q holds afterwards. To deal with non-local control flow and function calls, our judgments become sextuples $\Delta;\, J;\, R \vdash \{P\}\, s\, \{Q\}$, where:

- Δ is a finite function from function names to their pre- and postconditions. This environment is used to cope with (mutually) recursive functions.
- J is a function that gives the jumping condition for each `goto`. When executing a `goto` l, the assertion $J\, l$ has to hold.
- R is the assertion that has to hold when executing a `return`.

The assertions P, Q, J and R correspond to the four directions $\searrow$, $\nearrow$, $\curvearrowright$ and $\twoheaduparrow$ in which traversal through a statement can be performed. We therefore often treat the sextuple as a triple $\Delta;\, \bar{P} \vdash s$, where $\bar{P}$ is a function from directions to assertions such that $\bar{P} \searrow = P$, $\bar{P} \nearrow = Q$, $\bar{P}\, (\curvearrowright l) = J\, l$ and $\bar{P} \twoheaduparrow = R$.

We use a shallow embedding for the representation of assertions. This treatment is similar to that of Appel and Blazy [2] and Von Oheimb [15].

Definition 4.1. Assertions *are predicates over the the stack and the memory. We define the following connectives on assertions.*

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

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

Definition 4.2. *An assertion P is* stack independent *if $P\,\rho_1\,m \to P\,\rho_2\,m$ for each memory m and stacks ρ_1 and ρ_2, and similarly is* memory independent *if $P\,\rho\,m_1 \to P\,\rho\,m_2$ for each stack ρ and memories m_1 and m_2.*

Next, we define the assertions of separation logic [14]. The assertion emp asserts that the memory is empty. 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. Finally, $e_1 \mapsto e_2$ asserts that the memory consists of exactly one cell at address e_1 with contents e_2, and $e_1 \hookrightarrow e_2$ asserts that the memory contains at least a cell at address e_1 with contents e_2.

Definition 4.3. *The* connectives of separation logic *are defined as follows.*

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

To deal with block scope variables we need to lift an assertion such that the De Bruijn indexes of its variables are increased. We define the lifting $P\uparrow$ of an assertion P semantically, and prove that it indeed behaves as expected.

Definition 4.4. *The assertion $P\uparrow$ is defined as $\lambda\rho\,m\,.\,P\,(\mathsf{tail}\,\rho)\,m$.*

Lemma 4.5. *We have $(e \Downarrow v)\uparrow = (e\uparrow) \Downarrow v$ and $(e_1 \mapsto e_2)\uparrow = (e_1\uparrow) \mapsto (e_2\uparrow)$, where the operation $e\uparrow$ replaces each variable x_i in e by x_{i+1}. Furthermore, $(\_)\uparrow$ distributes over the connectives $\to$, $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, and $*$.*

In order to relate the pre- and postcondition of a function, we allow universal quantification over arbitrary logical variables $\vec{y}$. The specification of a function with parameters $\vec{v}$ consists therefore of a precondition $P\,\vec{y}\,\vec{v}$ and postcondition $Q\,\vec{y}\,\vec{v}$. These should be stack independent because local variables will have a different meaning at the calling function than in the called function's body. We will write such a specification as $\forall\vec{y}\forall\vec{v}\,.\,\{P\,\vec{y}\,\vec{v}\}\,\{Q\,\vec{y}\,\vec{v}\}$.

Definition 4.6. *Given a function δ assigning statements to function names, the rules of the axiomatic semantics are defined as:*

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

$$\frac{\begin{array}{c}(\forall l \in \mathsf{labels}\,s\,.\,J'l \to Jl) \quad (\forall l \notin \mathsf{labels}\,s\,.\,Jl \to J'l) \quad R \to R'\\ P' \to P \quad \Delta;\,J;\,R \vdash \{P\}\,s\,\{Q\} \quad Q \to Q'\end{array}}{\Delta;\,J';\,R' \vdash \{P'\}\,s\,\{Q'\}} \quad \text{(weaken)}$$

$$\frac{}{\Delta;\,J;\,R \vdash \{P\}\,\mathtt{skip}\,\{P\}} \qquad \frac{}{\Delta;\,J;\,R \vdash \{R\}\,\mathtt{return}\,\{Q\}} \quad \text{(skip \& return)}$$

$$\frac{}{\Delta;\, J;\, R \vdash \{\exists a\, v\,.\, e_1 \Downarrow a \wedge e_2 \Downarrow v \wedge \mathtt{ptr}\ a \hookrightarrow - \wedge P[a := v]\}\ e_1 := e_2\ \{P\}} \quad \text{(assign)}$$

$$\frac{\Delta;\, J;\, R \vdash \{J\ l\}\ s\ \{Q\}}{\Delta;\, J;\, R \vdash \{J\ l\}\ l : s\ \{Q\}} \qquad \frac{}{\Delta;\, J;\, R \vdash \{J\ l\}\ \mathtt{goto}\ l\ \{Q\}} \quad \text{(label \& goto)}$$

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

$$\frac{\Delta;\, J;\, R \vdash \{P\}\ s_1\ \{P'\} \quad \Delta;\, J;\, R \vdash \{P'\}\ s_2\ \{Q\}}{\Delta;\, J;\, R \vdash \{P\}\ s_1\,;\, s_2\ \{Q\}} \quad \text{(comp)}$$

$$\frac{\Delta;\, J;\, R \vdash \{e \Downarrow \top \wedge P\}\ s_1\ \{Q\} \quad \Delta;\, J;\, R \vdash \{e \Downarrow \bot \wedge P\}\ s_2\ \{Q\}}{\Delta;\, J;\, R \vdash \{e \Downarrow - \wedge P\}\ \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2\ \{Q\}} \quad \text{(cond)}$$

$$\frac{\Delta\, f = \{P\}\,\{Q\} \quad \vec{e} \Downarrow \vec{v} \wedge P\,\vec{y}\,\vec{v} \to A \quad A \textit{ memory independent}}{\Delta;\, J;\, R \vdash \{\vec{e} \Downarrow \vec{v} \wedge P\,\vec{y}\,\vec{v}\}\ f(\vec{e})\ \{A \wedge Q\,\vec{y}\,\vec{v}\}} \quad \text{(call)}$$

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

The traditional *frame rule* of separation logic [14] includes the side-condition $\mathsf{vars}\, s \cap \mathsf{vars}\, A = \emptyset$ on the free variables in the statement s and assertion A. However, as our local variables are just (immutable) references into the memory, we do not need this side-condition. Also, the (frame) and (block) rule are uniform in all assertions, allowing us to write:

$$\frac{\Delta;\, \bar{P} \vdash s}{\Delta;\, A * \bar{P} \vdash s} \qquad \frac{\Delta;\, \bar{P} \vdash \mathtt{block}\ s}{\Delta;\, x_0 \mapsto - * \bar{P}\uparrow \vdash s}$$

Since the return and goto statements leave the normal control flow, the post-conditions of the (goto) and (return) rules are arbitrary.

Our rules for function calls are similar to those by Von Oheimb [15]. The (call) rule is to call a function that is already in Δ. It is important to notice that its postcondition is not $\vec{e} \Downarrow \vec{v} \wedge Q\,\vec{y}\,\vec{v}$, as after calling f evaluation of $\vec{e}$ may be different after all. However, in case $\vec{e}$ contains no load expressions, we have that $\vec{e} \Downarrow \vec{v}$ is memory independent, and we can simply take A to be $\vec{e} \Downarrow \vec{v}$.

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 $\Pi_*[x_i \mapsto v_i] * P'\vec{y}\,\vec{v}$, where $\Pi_*[x_i \mapsto v_i]$ denotes the assertion $x_i \mapsto v_i * \ldots x_n \mapsto v_n$, states that the function parameters $\vec{x}$ are allocated with values $\vec{v}$ for which the precondition P' of the function holds. The post- and returning condition $\Pi_*[x_i \mapsto \_] * Q'\vec{y}\,\vec{v}$ ensure that the parameters have not been deallocated while executing the function body and that the postcondition P' of the function holds on a return. The jumping condition $\lambda l.\mathsf{False}$ ensures that all gotos jump to a label that occurs in the function body.

Euclid's algorithm in C:

```
void swap(int *p, int *q) {
  int z = *p; *p = *q; *q = z;
}

int gcd(int y, int z) {
  l: if (z) {
    y = y % z; swap(&y, &z); goto l;
  }
  return y;
}
```

Verification of the body of `swap`:

$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto y * q \mapsto z\}$
`block (`
$\{x_0 \mapsto - * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$
$x_0 :=$ `load (load` x_1`)`;
$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto y * q \mapsto z\}$
`load` $x_1 :=$ `load (load` x_2`)`;
$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto z * q \mapsto z\}$
`load` $x_2 :=$ `load` x_0
$\{x_0 \mapsto y * x_1 \mapsto p * x_2 \mapsto q * p \mapsto z * q \mapsto y\}$
`)`
$\{x_0 \mapsto p * x_1 \mapsto q * p \mapsto z * q \mapsto y\}$

Verification of the body of `gcd`:

$\{x_0 \mapsto$ `int` $y * x_1 \mapsto$ `int` $z\}$
l :
$\{J\ l\}$
`if (load` x_1`) (`
$\{$`load` $x_1 \Downarrow \top \wedge J\ l\}$
$\{x_0 \mapsto$ `int` $y' * x_1 \mapsto$ `int` $z' \wedge$
$z' \neq 0 \wedge$ `gcd` $y\ z =$ `gcd` $y'\ z'\}$
$x_0 :=$ `load` x_0 % `load` x_1 ;
$\{x_0 \mapsto$ `int` $(y'$ % $z') * x_1 \mapsto$ `int` $z' *$
$(z' \neq 0 \wedge$ `gcd` $y\ z =$ `gcd` $y'\ z' \wedge$ `emp`$)\}$
`swap(`x_0, x_1`)`;
$\{x_0 \mapsto$ `int` $z' * x_1 \mapsto$ `int` $(y'$ % $z') *$
$(z' \neq 0 \wedge$ `gcd` $y\ z =$ `gcd` $y'\ z' \wedge$ `emp`$)\}$
$\{J\ l\}$
`goto` l
$\{x_0 \mapsto$ `int (gcd` $y\ z) * x_1 \mapsto$ `int` $0\}$
`) else`
$\{$`load` $x_1 \Downarrow \bot \wedge J\ l\}$
$\{x_0 \mapsto$ `int` $y' * x_1 \mapsto$ `int` $0 \wedge$ `gcd` $y\ z =$ `gcd` $y'\ 0\}$
`skip`
$\{x_0 \mapsto$ `int (gcd` $y\ z) * x_1 \mapsto$ `int` $0\}$

Fig. 2. Verification of Euclid's algorithm.

As an example we verify Euclid's algorithm for computing the greatest common divisor. We first verify the `swap` function, which takes two pointers p and q and swaps their contents. Its specification is as follows:

$$\forall y\, z\, \forall p\, q\,.\, \{p \mapsto y * q \mapsto z\}\ \{p \mapsto z * q \mapsto y\}$$

Here, universal quantification over the logical variables y and z is used to relate the contents of the pointers p and q in the pre- and postcondition. In order to add this function to the context Δ of verified functions using the (add funs) rule, we have to prove that the body satisfies the above specification. An outline of this proof (with implicit uses of weakening) is displayed in Figure 2.

To verify the body of the `gcd` function, we use a jumping environment J that assigns $\exists y' z'\,.\, x_0 \mapsto$ `int` $y' * x_1 \mapsto$ `int` $z' \wedge$ `gcd` $y\ z =$ `gcd` $y'\ z'$ to the label l. For the function call to `swap`, we use the (frame) rule with the framing condition $z' \neq 0 \wedge$ `gcd` $y\ z =$ `gcd` $y'\ z' \wedge$ `emp` as displayed in Figure 2. We refer to the Coq formalization for the full details of these proofs.

5 Soundness of the axiomatic semantics

We define $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$ for Hoare sextuples in terms of our operational semantics. Proving *soundness* of the axiomatic semantics then consists of showing that $\Delta;\ J;\ R \vdash \{P\}\, s\, \{Q\}$ implies that $\Delta;\ J;\ R \vDash \{P\}\, s\, \{Q\}$.

We want $\Delta; J; R \vDash \{P\}\, s\, \{Q\}$ to ensure partial program correctness. Intuitively, that means: if $P\, k\, m$ and $\mathbf{S}(k, (s, \searrow), m) \twoheadrightarrow^* \mathbf{S}(k, (s, \nearrow), m')$, then $Q\, k\, m'$. However, due to the additional features, this is too simple.

1. We also have to account for reductions starting in $\Uparrow$ or $\curvearrowright$ direction. Hence, we take the four assertions J, R, P and Q together as one function $\bar{P}$ and write $\Delta; J; R \vDash \{P\}\, s\, \{Q\}$ as $\Delta; \bar{P} \vDash s$. The intuitive meaning of $\Delta; \bar{P} \vDash s$ is: if $\bar{P}\, d\, k\, m$ and $\mathbf{S}(k, (s, d), m) \twoheadrightarrow^* \mathbf{S}(k, (s, d'), m')$, then $\bar{P}\, d'\, k\, m'$.
2. We have to enforce the reduction $\mathbf{S}(k, (s, d), m) \twoheadrightarrow^* \mathbf{S}(k, (s, d'), m')$ to remain below k as it could otherwise take too many items off the context.
3. Our language has various kinds of undefined behavior (*e.g.* invalid pointer dereferences). We therefore also want $\Delta; \bar{P} \vDash s$ to guarantee that s does not exhibit such undefined behaviors. Hence, $\Delta; \bar{P} \vDash s$ should at least guarantee that if $\bar{P}\, d\, k\, m$ and $\mathbf{S}(k, (s, d), m) \twoheadrightarrow_k^* S$, then S is either:
   - reducible (no undefined behavior has occurred); or:
   - of the shape $\mathbf{S}(k, (s, d'), m')$ with $\bar{P}\, d'\, k\, m'$ (execution is finished).
4. The program should satisfy a form of memory safety so as the make the frame rule derivable. Hence, if before execution the memory can be extended with a disjoint part, that part should not be modified during the execution.
5. We take a step indexed approach in order to relate the assertions of functions in Δ to the statement s.

Together this leads to the following definitions:

Definition 5.1. *Given a predicate $\bar{P}$ over stacks, focuses and memories, specifying valid ending states, the relation $\bar{P} \vDash_n \hat{\mathbf{S}}(k, \phi, m \cup \square)$ is defined as: for each reduction $\mathbf{S}(k, \phi, m \cup m_f) \twoheadrightarrow_k^n \mathbf{S}(k', \phi', m')$, we have that m' is of the shape $m' = m'' \cup m_f$ for some memory m'', and either:*

1. *there is a state S such that $\mathbf{S}(k', \phi', m') \twoheadrightarrow_k S$; or:*
2. *$k' = k$ and $\bar{P}\, k'\, \phi'\, m''$.*

Definition 5.2. Validity of the environment Δ, *notation $\vDash_n \Delta$ is defined as: if $\Delta\, f = (\forall \vec{y}\,.\, \forall \vec{v}\,.\, \{P\, \vec{y}\, \vec{v}\}\, \{Q\, \vec{y}\, \vec{v}\})$ and $P\, \vec{y}\, \vec{v}\, k\, m$, then $Q' \vDash_n \hat{\mathbf{S}}(k, \overline{\mathsf{call}}\, f\, \vec{v}, m \cup \square)$, where $Q' := \lambda \rho\ \phi\ m'\,.\, (\phi = \overline{\mathsf{return}}) \wedge Q\, \vec{y}\, \vec{v}\, \rho\, m'$.*

Definition 5.3. Validity of a statement s, *notation $\Delta; \bar{P} \vDash s$ is defined as: if $\vDash_n \Delta$, $\mathsf{down}\, d\, s$, and $\bar{P}\, d\, k\, m$, then $Q' \vDash_n \hat{\mathbf{S}}(k, (d, s), m \cup \square)$, where*

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

The predicate down *holds if* $\mathsf{down} \searrow s'$ *or* $\mathsf{down}\ (\curvearrowright l)\ s'$ *with* $l \in \mathsf{labels}\ s'$.

Proposition 5.4 (Soundness). *$\Delta; \bar{P} \vdash s$ implies $\Delta; \bar{P} \vDash s$.*

This proposition is proven by induction on the derivation of $\Delta; \bar{P} \vdash s$. Thus, for each rule of the axiomatic semantics, we have to show that it holds in the model. The rules for the skip, return, assignment, goto and function calls are proven by chasing all possible reduction paths. In the case of the assignment statement, we need weakening of expression evaluation (Lemma 2.5).

All structural rules are proven by induction on the length of the reduction. These proofs involve chasing all possible reduction paths. We refer to the Coq formalization for the proofs of these rules.

6 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.

There are some small differences between the Coq development and this paper. Firstly, we omitted while statements and functions with return values here, whereas they are included in the Coq development. Secondly, in this paper, we presented the axiomatic semantics as an inference system, and then showed that it has a model. Since we did not consider completeness, in Coq we directly proved all rules to be derivable with respect to the model.

We used type classes to provide abstract interfaces for commonly used structures like finite sets and finite functions, so 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 [17]. However, whereas their work heavily relies on *setoids* (types equipped with an equivalence relation), we tried to avoid that, and used Leibniz equality wherever possible. In particular, our interface for finite functions requires *extensionality* with respect to Leibniz equality. That means $m_1 = m_2 \leftrightarrow \forall x \,.\, m_1\, x = m_2\, x$.

Intensional type theories like Coq do not satisfy extensionality. However, finite functions indexed by a countable type can still be implemented in a way that extensionality holds. For the memory we used finite functions indexed by binary natural numbers implemented as radix-2 search trees. This implementation is based on the implementation in CompCert [12]. But whereas CompCert's implementation does not satisfy canonicity, and thus allows different trees for the same finite function, we have equipped our trees with a proof of canonicity. This way, equality on these finite functions as trees becomes extensional.

Extensional equality on finite functions is particularly useful for dealing with assertions, which are defined as predicates on the stack and memory (Definition 4.1). Due to extensionality, we did not have to equip assertions with a proof of well-definedness with respect to extensional equality on memories.

Although the semantics described in this paper is not extremely big, it is still quite cumbersome to be treated without automation in a proof assistant. In particular, the operational semantics is defined as an inductive type consisting of 32 constructors. To this end, we have automated many steps of the proofs. For example, we implemented the tactic `do_cstep` to automatically perform reduction steps and to solve the required side-conditions, and the tactic `inv_cstep` to perform case analyzes on reductions and to automatically discharge impossible cases. 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 3500 lines of code including comments and white space. Apart from that, the library on general purpose theory (finite sets, finite functions, lists, *etc.*) is about 7000 lines, and the `gcd` example is about 250 lines.

7 Conclusions and further research

The further reaching goal of this work is to develop an operational semantics for a large part of the C11 programming language [8] as part of the Formalin project [10]. In order to get there, support for non-local control flow is a necessary step. The operational semantics in this paper extends easily to most other forms of non-local control in C: the break and continue statement, and non-structured switch statements (*e.g.* Duff's device). To support these, we just have to add an additional *direction* and its corresponding reduction rules.

In this paper we have also defined an axiomatic semantics. The purpose of this axiomatic semantics is twofold. Firstly, it gives us more confidence in the correctness and usability of our operational semantics. Secondly, in order to reason about actual C programs, non-local control flow and pointers to block scope variables have to be supported by the axiomatic semantics.

Unfortunately, the current version of our axiomatic semantics is a bit cumbersome to be used for actual program verification. The foremost reason is that our way of handling local variables introduces some overhead. In traditional separation logic [14], there is a strict separation between local variables and allocated storage: the values of local variables are stored directly on the stack, whereas the memory is only used for allocated storage. To that end, the separating conjunction does not deal with local variables, and many assertions can be written down in a shorter way. For example, even though we do not use pointers to the local variables of the `swap` function (Figure 2), we still have to deal with two levels of indirection.

It seems not too hard to make allocation of local variables in the memory optional, so that it can be used only for variables that actually have pointers to them. Ordinary variables then correspond nicely to those with the `register` keyword in C. Alternatively, the work of Parkinson *et al.* [16] on variables as resources may be useful.

Another requirement to conveniently use an axiomatic semantics for program verification is strong automation. Specific to the Coq proof assistant there has been work on this by for example Appel [1] and Chlipala [3]. As our main purpose is to develop an operational semantics for a large part of C11, we consider automation a problem for future work.

In order to get closer to a semantics for C11 we are currently investigating the following additional features of the C11 standard.

- Expressions with side effects and *sequence points*.
- The C type system including structs, unions, arrays and integer types.
- The non-aliasing restrictions (*effective types* in particular).

We intend to support these features in both our operational and axiomatic semantics. Ongoing work shows that our current operational semantics can easily be extended with non-deterministic expressions using a similar approach as Norrish [13] and Leroy [12]. As non-determinism in expressions is closely related to concurrency, we use separation logic for a Hoare logic for expressions.

In this paper we have not considered completeness of the axiomatic semantics as it is not essential for program verification. Also, our future extension for non-deterministic expressions with side-effects will likely be incomplete.

Another direction for future research is to relate our semantics to the CompCert semantics [12] (by eliminating block scope variables). That way we can link it to actual non-local jumps in assembly.

Acknowledgments. We thank Erik Poll for bringing up the idea of an axiomatic semantics for gotos, and thank Herman Geuvers and the anonymous referees for providing several helpful suggestions. 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. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011.
4. E. W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968. Letter to the Editor.
5. C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.
6. M. Felleisen and R. Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992.
7. G. P. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
8. International Organization for Standardization. *ISO/IEC 9899-2011: Programming languages – C*. ISO Working Group 14, 2012.
9. D. Knuth. Structured programming with go to statements. In *Classics in software engineering*, pages 257–321. Yourdon Press, 1979.
10. 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.
11. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
12. X. Leroy. The CompCert verified compiler, software and commented proof. Available at `http://compcert.inria.fr/`, 2012.
13. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
14. 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.
15. D. v. Oheimb. Hoare Logic for Mutual Recursion and Local Variables. In *FSTTCS*, volume 1738 of *LNCS*, pages 168–180, 1999.
16. M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as Resource in Hoare Logics. In *LICS*, pages 137–146, 2006.
17. B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.
18. H. Tews. Verifying Duff's device: A simple compositional denotational semantics for Goto and computed jumps, 2004.