A framework for describing recursive data structure topologies in Coq

Kenneth Roe and Scott Smith

The Johns Hopkins University

Abstract. This paper presents an axiomatic framework in Coq for verifying invariants on heap data structures such as lists and trees in a C-like language with a low-level store model. The goal of the framework is to detect common errors such as memory leaks, dangling pointers and looped data structures. The framework provides a language for expressing invariants, and a set of inference axioms for verifying them on code that manipulates the data structures. This work builds on the work done by Cook et al. which uses separation logic with recursive predicates to document data structure invariants. The key extension here is the ability to express and reason about data structures more complex than linked lists. The heap description includes a spatial component describing the basic set of lists and trees in the heap. New logical constructs are included to describe special pointer field invariants such as back pointers. We use the framework to formally prove in Coq the heap invariants of a small example program that generates a linked list representing the tree traversal of a tree. This proof guarantees the integrity of program's data structures and that common errors such as memory leaks or dangling pointer references did not arise. We define the meaning of the abstract state in terms of a simpler concrete state. We also include a number of axioms for reasoning about the abstract state that are used in the Coq verification. Proving soundness of these abstract axioms in Coq remains future work.

1 Introduction

Within the last few years, shape analysis has begun to make headway in verifying complex data structures[5, 16, 4, 15]. The most successful of these approaches involves the use of separation logic [14] extended to handle recursive data structures [5]. In the aforecited work, a language for abstract description of heaps is defined which can be used to specify the set of linked lists the heap contains, and a set of cells also contained in the heap but not connected to any of the linked lists. An enhanced version of this work [8] in which doubly linked lists can also be represented was used to discover memory problems in Microsoft drivers. There are also several efforts in implementing separation logic frameworks in theorem proving environments, for example [13, 12, 1, 17, 11, 2].

This paper describes a methodology for expanding the framework of [5] to cover more complex data structures including trees and doubly linked lists. The

formalism in this paper adds considerably to the flexibility of the types of data structures that can be represented. In addition to handling back pointers as in [7], the framework can handle pointers that go from one data structure into another completely independent data structure. Also, our methodology provides a good mechanism for representing information about pointer variables which iterate through a data structure. To achieve this increased expressiveness, we use an abstract state that contains a spatial component which is similar to earlier systems, as well as a more complex predicate component in which an extension of first order logic can be used to specify heap properties.

We have implemented our framework in Coq[3] and constructed a complete formal proof of correctness for a small tree traversal program which we describe in this paper. The proof verifies that the program maintains the integrity of its data structures and does not have errors such as memory leaks or dangling pointer references. We have developed an abstract state representation which is an extension of the representation used in [5]. The Coq formalization incorporates Hoare axioms to systematically back-propagate the abstract state through the program. We have also developed folding and unfolding axioms for the recursive spatial heap predicates of separation logic. They are needed to handle join points in processing if statements as well as setting up the application of the axiom used to back propagate over a heap store. While the formal proof of the particular example is complete, the task of proving all the heap logic axioms used in the proof to be sound in Coq is a challenging task and remains future work.

The structure of the remainder of the paper is as follows. In Section 2 we present the example program and informally describe properties of the program that our formal Coq proof verifies. In Section 3 we present the abstract heap state representation, followed by Section 4 which contains the programming language syntax and semantics as well as axioms for reasoning about abstract heap states under program execution. In Section 5 we outline how portions of the proof for the example program were performed in Coq.

2 An example and the properties we proved

Figure 1 shows a program which performs a traversal of a tree and places the result in a linked list, pointed at by the global variable `p`. Our Coq proof verifies a number of important properties of the data structures in this program. Some properties that can be read off of the heap invariant from our formal proof include the following.

- The program maintains two well formed linked lists, the heads of which are pointed to by `n` and `p`. By well formed we mean that memory on the heap is properly allocated for the lists and there are no loops in the data structures.
- The program maintains a well formed tree pointed to by `r`.
- `t` always points to an element in the tree rooted at `r`.
- The two lists and the tree do not share any nodes.

```
struct list {
 struct list *n;
 struct tree *t;
 int data;
};

struct tree {
 struct tree *l, *r;
 int value;
};
struct list *p;
void build_pre_order(struct tree *r) {
ℓ1  struct list *i = NULL, *n, *x;
ℓ2  struct tree *t = r;
ℓ3  p = NULL;
ℓ4  while (t) {
ℓ5   n = p;
ℓ6   p = malloc(sizeof(struct list));
ℓ7   p->t = t;
ℓ8   p->n = n;
ℓ9   if (t->l==NULL && t->r==NULL) {
ℓ10    if (i==NULL) {
}
ℓ11     t = NULL;
ℓ12    } else {
ℓ13     struct list *tmp = i->n;
ℓ14     t = i->t;
ℓ15     free(l);
ℓ16     i = tmp;
ℓ17    }
ℓ18   } else if (t->r==NULL) {
ℓ19    t = t->l;
ℓ20   } else if (t->l==NULL) {
ℓ21    t = t->r;
ℓ22   } else {
ℓ23    n = i;
ℓ24    i = malloc(
         sizeof(struct list));
ℓ25    i->n = n;
ℓ26    x = t->r;
ℓ27    i->t = x;
ℓ28    t = t->l;
ℓ29   }
ℓ30  }
ℓ31 }
```

Fig. 1. Example program which builds a linked list of all nodes in a tree

- Other than the memory used for the two lists and the tree, no other heap memory is allocated.
- the `t` field of every element in both list structures points to an element in the tree.

We note that some of these properties cannot be verified by existing separation logic frameworks, most notably the last one.

3 Abstract and concrete heap domains

In this section we define the concrete and abstract heaps, and define how a particular concrete (actual runtime) heap may satisfy a given abstract heap specification. We adapt the abstract and concrete heap domain definitions from [5]. For the concrete heap we use $Stack \triangleq (Var \cup Var') \rightarrow Val$, $Heap \triangleq Val - \texttt{NULL} \rightharpoonup_{fin} (Fld \rightharpoonup Val)$ and $States \triangleq Stack \times Heap$. *Val* is an integer that can be treated as either a numeric value or a pointer. `NULL` is a synonym for the integer 0. We explicitly exclude `NULL` from the domain of *Heap*. The cells are slightly different than those used by [5]. Rather than being a single value, they are a mapping from field names to values. Hence, the cons cell for a list or a node in a tree is a single cell in our representation.

For the abstract domain, we adapt [5]'s symbolic heaps. We replace their `ls` list operator with a $\mathcal{R}$ operator for expressing the structure of more general recursive data structures. We have expanded their Π expressions to include a path operator and quantifiers in addition to equality. Both Π and Σ have been extended to allow simple arithmetic expressions to occur at various points in formulae. The grammar is given in Figure 2. More details on the novel syntax will be given below when their meaning is defined.

v, x, y, z	$\in \mathit{Var}$	integer variables
v', x', y', z'	$\in \mathit{Var}'$	primed variables
$\mathtt{f}$	$\in \mathit{Fld}$	fields
F	$\in \mathcal{P}(\mathit{Fld})$	sets of fields
X	$::= x \mid x'$	
E	$::= X \mid E + E \mid E - E \mid E * E \mid \mathit{Val}$	expressions
Π	$::= \mathit{Body} \mid X \in \Sigma \wedge \Pi$	
Body	$::= \mathtt{true} \mid E = E \mid \mathit{Body} \wedge \mathit{Body} \mid \mathit{Body} \vee \mathit{Body}$	
	$\mid \forall \mathit{Var} \in \Sigma.\mathit{Body} \mid \exists \mathit{Var} \in \Sigma.\mathit{Body} \mid \neg \mathit{Body} \mid E \overset{\mathtt{f}}{\rightsquigarrow} E$	pure formulae
Σ	$::= \mathtt{emp} \mid \Sigma * \Sigma \mid \Sigma -\!\!* \Sigma \mid X \mapsto \{\overline{\mathtt{f} : X}\}$	
	$\mid \mathcal{R}_F(X, \overline{X}) \mid \mathtt{junk}$	Spatial heaps
H	$::= \Pi \mid \Sigma$	Symbolic heaps

Fig. 2. Grammar for concrete and abstract heaps

A symbolic heap is defined to be valid in a given concrete heap via the relation $s, h \vdash \Pi|\Sigma$, where $s \in \mathit{Stack}$, $h \in \mathit{Heap}$, and $\Pi|\Sigma$ is a symbolic heap. This relationship defines the set of concrete states, s, h which are represented by the abstract state $\Pi|\Sigma$. Through the rest of this section, we will build up the definition of $s, h \vdash \Pi|\Sigma$. Note that formulae are considered well-formed only if all primed variables (v', x', y' and z') are existentially quantified; we may sometimes informally leave off such a quantifier but it is implicitly there.

3.1 The semantics of $s, h \vdash \Sigma$

In this section we give semantics for symbolic heaps, covering each Σ grammar component in turn.

We first introduce a number of auxiliary definitions. Note that throughout this section we will use $h \vdash \Sigma$ to abbreviate there exists an s such that $s, h \vdash \mathtt{true}|\Sigma$ and $s, h \vdash \Sigma$ to abbreviate $s, h \vdash \mathtt{true}|\Sigma$.

Implicit existential quantification of primed variables Throughout the remainder of the paper $s, h \vdash \Pi|\Sigma$ should be interpreted as there exists an s' which is an extension of s mapping all primed variables in $\Pi|\Sigma$ such that $s', h \vdash \Pi|\Sigma$.

Separation logic constructs The $*$, $-\!\!*$ and $\mapsto$ operators are adapted from separation logic [14] and we assume familiarity with them here. Our heap model is slightly different since the heap locations contain a mapping from field identifiers to values rather than just integers. The * and $-\!\!*$ operators are standard.

Definition 1. *$s, h \vdash \Sigma_1 * \Sigma_2$ if and only if there exists h_1 and h_2 such that $h = h_1 \cup h_2$, $\mathtt{dom}(h_1) \cap \mathtt{dom}(h_2) = \emptyset$, $s, h_1 \vdash \Sigma_1$ and $s, h_2 \vdash \Sigma_2$.*

Definition 2. *$s, h \vdash \Sigma_1 - \!\!* \Sigma_2$ if and only if there exists h and h_2 such that $h = h_1 \cup h_2$, $\mathtt{dom}(h_1) \cap \mathtt{dom}(h_2) = \emptyset$, $s, h \vdash \Sigma_1$ and $s, h_2 \vdash \Sigma_2$.*

Singleton cell constructs We use atomic assertion $x \mapsto \{\overline{\mathtt{f} : x}\}$ to indicate a singleton cell mapping in the abstract state. Its meaning is defined as follows.

Definition 3. *$s, h \vdash l \mapsto \{\mathtt{f}_1 : v_1, \ldots, \mathtt{f}_n : v_n\}$ if and only if $h = \{s(l) \mapsto c\}$ such that $\{\mathtt{f}_1 \mapsto s(v_1), \ldots, \mathtt{f}_n \mapsto s(v_n)\} \subseteq c$.*

Recursive data structures The $\mathcal{R}_{\mathtt{F}}(x, \overline{X})$ predicate is used to represent recursive data structures in the spatial heap. These data structures can be either lists or trees. The x parameter is the root of the data structure. The subscripted F is the set of fields used for recursive descent. For a list, this could for example be the field labelled `next`. For a tree, there are usually two fields `left` and `right` which are pointers to the left and right subtree. The third parameter represents early stopping points in the recursion. It is used in the representation of states where the modification of a data structure is in progress. For example, consider a program that frees a leaf node but has not as yet updated the pointer from the parent to that node. Before the parent is updated, it needs to be marked as an early stopping point of the recursion.

We define the meaning of $\mathcal{R}$ by induction on the concrete heap.

Definition 4. *The meaning of $\mathcal{R}_F(x, \overline{X})$ in a concrete state s, h, where $x \in \mathtt{dom}(s)$, is defined to be the least relation that satisfies the following clauses:*

$s, h \vdash \mathcal{R}_{\mathtt{F}}(x, \overline{X})$ *iff* $s, h \vdash \mathtt{emp}$ *where* $x \in \overline{X}$

$s, h \vdash \mathcal{R}_{\mathtt{F}}(x, \overline{X})$ *iff* $s, h \vdash \mathtt{emp}$ *where* $s(x) = \mathtt{NULL}$

$s, h \vdash \mathcal{R}_{\mathtt{f}_1,\ldots,\mathtt{f}_n}(x, \overline{X})$ *iff* *where* $x \notin \overline{X} \wedge s(x) \neq \mathtt{NULL}$

$$\exists y_1, \ldots, y_n, h_0, \ldots h_n.$$
$$s, h_0 \vdash l \mapsto \{\mathtt{f}_1 \mapsto y_1, \ldots, \mathtt{f}_n \mapsto y_n\} \wedge$$
$$s, h_1 \vdash \mathcal{R}_{\mathtt{f}_1,\ldots,\mathtt{f}_n}(y_1, \overline{X}) \wedge \ldots \wedge s, h_n \vdash \mathcal{R}_{\mathtt{f}_1,\ldots,\mathtt{f}_n}(y_n, \overline{X}) \wedge$$
$$h_0 \cup \ldots \cup h_n = h \wedge \forall i, j . 0 \leq i, j \leq n \wedge i \neq j \rightarrow h_i \cap h_j = \emptyset$$

Often we will abbreviate $\mathcal{R}_{\mathtt{F}}(v, \{\})$ as $\mathcal{R}_{\mathtt{F}}(v)$.

Some examples of $\mathcal{R}$ To illustrate the use of $\mathcal{R}$ in characterizing recursive heap structures, we show some sample models that sometimes satisfy $\mathcal{R}$. We start with a simple linked list heap abstraction. In this abstraction, the list is rooted at x and nodes are connected together by the `next` field – this is expressed as $\mathcal{R}_{\texttt{next}}(x, \emptyset)$. The list in this first concrete heap example has two elements at locations x and y.

$$\{x \mapsto 1, y \mapsto 2\}, \{1 \mapsto (\texttt{next} : 2), 2 \mapsto (\texttt{next} : \texttt{NULL})\} \vdash \mathcal{R}_{\texttt{next}}(x, \emptyset)$$

The next example is a heap abstraction that does not model this $\mathcal{R}$ because there is a loop.

$$\{x \mapsto 1, y \mapsto 2\}, \{1 \mapsto (\texttt{next} : 2), 2 \mapsto (\texttt{next} : 1)\} \nvdash \mathcal{R}_{\texttt{next}}(x, \emptyset)$$

Another failure case is where the `NULL` pointer terminating the list fails to appear in the abstraction. Note that $2 \mapsto ()$ represents a cell in which no fields are defined; this should not be confused with an unallocated cell in the formalism.

$$\{x \mapsto 1, y \mapsto 2\}, \{1 \mapsto (\texttt{next} : 2), 2 \mapsto ()\} \nvdash \mathcal{R}_{\texttt{next}}(x, \emptyset)$$

It is still possible to partially characterize a heap in this kind of situation. In the following example we put z in the set of terminators for $\mathcal{R}$ and hence we do not traverse past it.

$$\{x \mapsto 1, y \mapsto 2, z \mapsto 3\}, \{1 \mapsto (\texttt{next} : 2), 2 \mapsto (\texttt{next} : 3), 3 \mapsto ()\} \vdash \mathcal{R}_{\texttt{next}}(x, \{z\})$$

We now have all the pieces to define $s, h \vdash \Sigma | \texttt{true}$ for all cases of Σ.

Definition 5. *We say that $s, h \vdash \Sigma$ if and only if Σ is*

`emp`	*and* $\texttt{dom}(h) = \emptyset$
`junk`	*and* $\texttt{dom}(h) \neq \emptyset$
$\Sigma_1 * \Sigma_2$	*and definition 1 holds*
$\Sigma_1 -\!\!* \Sigma_2$	*and definition 2 holds*
$x \mapsto \{\texttt{f}_1 \mapsto v_1 \ldots \texttt{f}_n \mapsto v_n\}$	*and definition 3 holds*
$\mathcal{R}_F(x, \overline{X})$	*and definition 4 holds*

3.2 The semantics of Π

Now we need to define $s, h \vdash \Pi | \Sigma$. We define each form of Π formulae in turn.

x* ∈ *Σ We use this construct to identify which portion of the heap contains a variable. Often a pointer variable is used to traverse a single data structure. This construct is useful in identifying which structure that variable belongs to. Also, this construct has a special case allowing the variable to be `NULL`. This is because most programs allow pointer variables to either point to a valid element in a structure or to be `NULL`. It is defined as follows.

Definition 6. *$s, h \vdash \Sigma_1 * \Sigma_2 | x \in \Sigma_1 \wedge \Pi$ if and only if either $x =$ `NULL` or $s(x) \in dom(h_1)$ where $s, h_1 \vdash \Sigma_1$ and $h_1 \subseteq h$ and $s, h \vdash \Pi | \Sigma_1 * \Sigma_2$.*

$\forall \boldsymbol{v \in \Sigma.Body}$ **and** $\exists \boldsymbol{v \in \Sigma.Body}$ These are the standard universal and existential quantifiers. However the variable v is restricted to the specified portion of Σ. Formally,

Definition 7. *$s, h \vdash \forall v \in \Sigma.Body|\Sigma * \Sigma'$ holds if and only if $s, h \vdash Body[v \mapsto n]|\Sigma * \Sigma'$ holds for all $n \in dom(h_1)$ where $h_1 \vdash \Sigma$ and $h_1 \subseteq h$ and $h \vdash \Sigma * \Sigma'$.*

and

Definition 8. *$s, h \vdash \exists v \in \Sigma.Body|\Sigma * \Sigma'$ holds if and only if there exists an n such that $s, h \vdash Body[v \mapsto n]|\Sigma * \Sigma''$ holds and $n \in dom(h_1)$ where $h_1 \vdash \Sigma$ and $h_1 \subseteq h$ and $h \vdash \Sigma * \Sigma'$.*

Expression evaluation We now define the meaning of simple arithmetic expressions that may appear in many of the atomic formulae, defining an $\mathcal{E}$ operator that gives the meaning of these arithmetic expressions.

Definition 9. *$\mathcal{E}_s(E)$ where E is an expression and $s \in Stack$ is defined as follows:*

$$\begin{array}{lll} \mathcal{E}_s(E_1 + E_2) & = \mathcal{E}_s(E_1) + \mathcal{E}_s(E_2) & \\ \mathcal{E}_s(E_1 - E_2) & = \mathcal{E}_s(E_1) - \mathcal{E}_s(E_2) & \\ \mathcal{E}_s(E_1 * E_2) & = \mathcal{E}_s(E_1) * \mathcal{E}_s(E_2) & \\ \mathcal{E}_s(l) & = l & \text{where } l \in Val \\ \mathcal{E}_s(x) & = s(x) & \text{where } x \in Var \end{array}$$

The $\boldsymbol{E \overset{\mathtt{f}}{\rightsquigarrow} E'}$ predicate This atomic formula asserts that dereferencing e via field `f` yields heap location e'. Its meaning is defined as follows:

Definition 10. *$h, s \vdash E \overset{\mathtt{f}}{\rightsquigarrow} E'$ holds if and only if $h(\mathcal{E}_s(E))\mathtt{f} = \mathcal{E}_s(E')$*

With the above definitions we can put everything together.

Definition 11. *We say that $s, h \vdash \Pi|\Sigma$ if and only if $s, h \vdash \Sigma$ and recursively if Π is*

$$\begin{array}{ll} x \in \Sigma \wedge \Pi_1 & \text{and definition 6 holds} \\ \Pi_1 \wedge \Pi_2 & \text{and } s, h \vdash \Pi_1|\Sigma \text{ and } s, h \vdash \Pi_2|\Sigma \\ \Pi_1 \vee \Pi_2 & \text{and } s, h \vdash \Pi_1|\Sigma \text{ or } s, h \vdash \Pi_2|\Sigma \\ \neg\Pi & \text{and } s, h \nvdash \Pi|\Sigma \\ \forall Var \in \Sigma.\Pi & \text{and definition 7 holds} \\ \exists Var \in \Sigma.\Pi & \text{and definition 8 holds} \\ E_1 = E_2 & \text{and } \mathcal{E}_s(E_1) = \mathcal{E}_s(E_2) \\ x \overset{\mathtt{f}}{\rightsquigarrow} y & \text{and definition 10 holds} \\ \mathtt{true} & \end{array}$$

4 Semantics of Programs

We now give semantics for a small imperative programming language, first via a definition of evaluation with respect to a concrete heap, and then by a set of principles axiomatizing program execution with respect to an abstract heap.

The grammar for the programming language is similar to that of [5]. However, since our locations can hold an entire structure rather than a single value, we need to introduce a field dereferencing operator, $E \to \mathtt{f}$. We give semantics for a language with the following grammar:

$b ::= E = E \mid E \neq E$
$p ::= x := E \mid x := E \to \mathtt{f} \mid E \to \mathtt{f} := x \mid x := \mathtt{new}(\overline{\mathtt{f}}) \mid \mathtt{delete}(E)$
$c ::= \mathtt{skip} \mid p \mid c; c \mid \mathtt{if}\ b\ \mathtt{then}\ c\ \mathtt{else}\ c \mid \mathtt{while}\ b\ \mathtt{do}\ c$

4.1 Concrete semantics

We now define a relation which represents a single step of execution in the concrete heap.

Definition 12. *We define the relation* $s, h \xrightarrow{c} s', h'$ *as follows:*

- $s, h \xrightarrow{\mathtt{skip}} s, h$
- $s, h \xrightarrow{x:=E} s[x \to \mathcal{E}_s(E)], h$
- *if* $l \notin dom(h)$ *and* $l \neq 0$ *then* $s, h \xrightarrow{x:=\mathtt{new}} s[x \to l], h[l \to \{\ldots\}]$
- *if* $l \in dom(h)$ *then* $s, h \xrightarrow{\mathtt{delete}\ l} s, h - \{l \to \ldots\}$
- *if* $\mathcal{E}_s(E) \in dom(h)$ *and* $\mathtt{f} \in dom(h(\mathcal{E}_s(E)))$ *then* $s, h \xrightarrow{\mathtt{x}:=E\to\mathtt{f}} s[x \to h(\mathcal{E}_s(E))\mathtt{f}], h$
- *if* $\mathcal{E}_s(E_1) \in dom(h)$ *then* $s, h \xrightarrow{E_1\to\mathtt{f}=E_2} s, h[\mathcal{E}_s(E_1) \to h(\mathcal{E}_s(E_1))[\mathtt{f} \to \mathcal{E}_s(E_2)]$
- *if* $s, h \xrightarrow{c_1} s', h'$ *and* $s', h' \xrightarrow{c_2} s'', h''$ *then* $s, h \xrightarrow{c_1;c_2} s'', h''$.
- *if* $\mathcal{E}_s(E_1) = \mathcal{E}_s(E_2)$ *and* $s, h \xrightarrow{c_1} s', h'$ *then* $s, h \xrightarrow{\mathtt{if}\ E_1=E_2\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2} s', h'$
- *if* $\mathcal{E}_s(E_1) \neq \mathcal{E}_s(E_2)$ *and* $s, h \xrightarrow{c_2} s', h'$ *then* $s, h \xrightarrow{\mathtt{if}\ E_1=E_2\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2} s', h'$
- *if* $\mathcal{E}_s(E_1) \neq \mathcal{E}_s(E_2)$ *and* $s, h \xrightarrow{c_1} s', h'$ *then* $s, h \xrightarrow{\mathtt{if}\ E_1\neq\ E_2\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2} s', h'$
- *if* $\mathcal{E}_s(E_1) \neq \mathcal{E}_s(E_2)$ *and* $s, h \xrightarrow{c_2} s', h'$ *then* $s, h \xrightarrow{\mathtt{if}\ E_1\neq E_2\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2} s', h'$
- *if* $\mathcal{E}_s(E_1) \neq \mathcal{E}_s(E_2)$ *then* $s, h \xrightarrow{\mathtt{while}\ E_1=E_2\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2} s, h$
- *if* $\mathcal{E}_s(E_1) = \mathcal{E}_s(E_2)$ *then* $s, h \xrightarrow{\mathtt{while}\ E_1\neq E_2\ \mathtt{do}\ c} s, h$
- *if* $\mathcal{E}_s(E_1) = \mathcal{E}_s(E_2)$ *and* $s, h \xrightarrow{c} s'h'$ *and* $s', h' \xrightarrow{\mathtt{while}\ E_1=E_2\ \mathtt{do}\ c} s'', h''$ *then* $s, h \xrightarrow{\mathtt{while}\ E_1=E_2\ \mathtt{do}\ c} s'', h''$
- *if* $\mathcal{E}_s(E_1) \neq \mathcal{E}_s(E_2)$ *and* $s, h \xrightarrow{c} s'h'$ *and* $s', h' \xrightarrow{\mathtt{while}\ E_1\neq E_2\ \mathtt{do}\ c} s'', h''$ *then* $s, h \xrightarrow{\mathtt{while}\ E_1\neq E_2\ \mathtt{do}\ c} s'', h''$

4.2 Abstract semantics

With the meaning of a concrete step defined, we can easily define when a step soundly takes one abstract heap to another.

Definition 13. $\Pi|\Sigma \xRightarrow{c} \Pi'|\Sigma'$ *iff for all* s, h *such that* $s, h \vdash \Pi|\Sigma$ *and* $s, h \xrightarrow{c} s', h'$ *for some* s', h'*, it follows that* $s', h' \vdash \Pi'|\Sigma'$.

In order to do symbolic program analysis, a number of rules need to be introduced. The rules are all designed to facilitate backward chaining. If for some step $\Pi|\Sigma \overset{c}{\Longrightarrow} \Pi'|\Sigma'$, a symbolic expression already exists for $\Pi'|\Sigma'$, the rules will make it easy to generate an expression for $\Pi|\Sigma$ so that the relationship holds. We do not necessarily guarantee generation of the weakest pre-condition. We do, however, generate a pre-condition that contains enough useful information for program verification. Each rule defines how to transform the abstract state for a specific statement. Note that the rules have not yet been proved sound with respect to the semantics – this is an axiomatic programming logic. Many of the rules are in fact very easy to prove sound, but a few are quite difficult and we leave overall soundness to future work.

While, if, sequence and skip We use the following rules to reason about these statements:

$$\frac{\Pi_1|\Sigma_1 \wedge b \overset{c_1}{\Longrightarrow} \Pi'|\Sigma' \qquad \Pi_2|\Sigma_2 \wedge \neg b \overset{c_2}{\Longrightarrow} \Pi'|\Sigma' \qquad \Pi|\Sigma \in join(\Pi_1|\Sigma_1, \Pi_2|\Sigma_2)}{\Pi|\Sigma \overset{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2}{\Longrightarrow} \Pi'|\Sigma'}(\texttt{absIf})$$

$$\frac{\Sigma^{Inv}|\Pi^{Inv} \overset{c}{\Longrightarrow} \Sigma^{Inv}|\Pi^{Inv} \qquad \Pi|\Sigma \wedge b \Longrightarrow \Sigma^{Inv}|\Pi^{Inv} \qquad \Sigma^{Inv}|\Pi^{Inv} \wedge \neg b \Longrightarrow \Pi'|\Sigma'}{\Pi|\Sigma \overset{\texttt{while } b \texttt{ do } c}{\Longrightarrow} \Pi'|\Sigma' \wedge \neg b}(\texttt{absWhile})$$

$$\frac{}{\Pi|\Sigma \overset{\texttt{skip}}{\Longrightarrow} \Pi|\Sigma}(\texttt{absSkip})$$

$$\frac{\Pi|\Sigma \overset{c_1}{\Longrightarrow} \Pi'|\Sigma' \quad \Pi'|\Sigma' \overset{c_2}{\Longrightarrow} \Pi''|\Sigma''}{\Pi|\Sigma \overset{c_1;c_2}{\Longrightarrow} \Pi''|\Sigma''}(\texttt{absSeq})$$

In the first two definitions above, b is used both in the program code and the predicate. When applying this rule, the AST of the b in the program code is mapped to a corresponding AST using the constructs in Π. $\Sigma^{Inv}|\Pi^{Inv}$ is the loop invariant for the while rule. In the first rule, used for processing if statements, generally, one is using them in a backchaining fashion. The $\Pi|\Sigma$ precondition needs to be generated as a join of $\Pi_1|\Sigma_1$ and $\Pi_2|\Sigma_2$. It is defined formally as

Definition 14. *$\Pi|\Sigma \in join(\Pi_1|\Sigma_1, \Pi_2|\Sigma_2)$ if and only if for all s, h it is the case that $s, h \vdash \Pi|\Sigma$ implies $s, h \vdash \Pi_1|\Sigma_1$ and $s, h \vdash \Pi|\Sigma$ implies $s, h \vdash \Pi_2|\Sigma_2$*

We discuss how to do this join in section 4.4.

Assignment, allocation and deallocation The following rules allow for reasoning about assignment, allocation and deallocation commands. The assignment operation is divided into three cases. The first case is where no heap information is involved. The second is an assignment of the form $v := E \rightarrow \texttt{f}$ which

involves retrieving a single value from the heap. The third is an assignment of the form $v \to \mathtt{f} := E$ where a single value is stored back to the heap. Unlike C or other popular programming languages we do not allow assignments that perform multiple stores or retrievals of values. This is to simplify the presentation; many common preprocessors such as CIL [6] can break down assignments into their atomic store components.

$\{\Pi[x/v] \wedge x = E | \Sigma[x/v]\} \overset{v:=E}{\Longrightarrow} \{\Pi | \Sigma\}$ (as1)

$\{v \in \Sigma' \wedge x' \in \Sigma' \wedge \Pi[x'/v] \wedge E \overset{\mathtt{f}}{\rightsquigarrow} x' | \Sigma[x'/v]\} \overset{v:=E \to \mathtt{f}}{\Longrightarrow} \{v \in \Sigma' \wedge \Pi | \Sigma\}$ (as2)

$\{\Pi' | \Sigma * x \mapsto \{\overline{\mathtt{f} : v}\}\} \overset{x \to \mathtt{f} := G}{\Longrightarrow} \{\Pi | \Sigma * x \mapsto \{\mathtt{f} \mapsto G, \overline{\mathtt{f} : v}\}\}$ (as3)
$\Pi' = \mathcal{T}(\Pi, \Sigma, x \mapsto \{\mathtt{f} : G, \overline{\mathtt{f} : v}\}, x \mapsto \{\overline{\mathtt{f} : v}\})$

$\{\Pi | \Sigma\} \overset{x:=\mathtt{new}}{\Longrightarrow} \{(\Pi | \Sigma) * x \mapsto \{\}\}$ (as4)
x not free in Π or Σ

$\{\Pi' | \Sigma * E \mapsto \{\overline{\mathtt{f} : v}\}\} \overset{\mathtt{delete}(E)}{\Longrightarrow} \{(\Pi | \Sigma)\}$ (as5)

The assignment rule `as3` requires Π be transformed to Π' via a $\mathcal{T}$ operator, defined below. Since the assignment occurring in this rule alters the heap, it changes the meaning of atomic formulae $l_1 \overset{\mathtt{f}}{\rightsquigarrow} l_2$ in Π and $\mathcal{T}$ updates any such atomic assertions affected by this particular assignment. Note that $\mathcal{T}$ is not a total function; in cases where it is not defined, the rule cannot apply.

Before defining $\mathcal{T}$, we need to establish two Lemmas. Many of the $l_1 \overset{\mathtt{f}}{\rightsquigarrow} l_2$ in Π are in a context where l_1 will never refer to the cell being modified. The following lemma allows us to simply carry these predicates from Π to Π'.

Lemma 1. *if* $s, h_1 \vdash \Sigma_1$, $s, h_2 \vdash \Sigma_2$ *and* $s, h_2' \vdash \Sigma_2'$ *then* $s, h_1 * h_2 \vdash l_1 \in \Sigma_1 \wedge l_1 \overset{\mathtt{f}}{\rightsquigarrow} l_2 | \Sigma_1 * \Sigma_2$ *if and only if* $s, h_1 * h_2' \vdash l_1 \in \Sigma_1 \wedge l_1 \overset{\mathtt{f}}{\rightsquigarrow} l_2 | \Sigma_1 * \Sigma_2'$

Basically, the above Lemma states that if we know that l_1 is outside of the portion of the heap being modified that the truth of the $l_1 \overset{\mathtt{f}}{\rightsquigarrow} l_2$ predicate remains unchanged.

For `as3`, there is a special case where we may have a predicate of the form $x \overset{\mathtt{f}'}{\rightsquigarrow} l_2$ in Π. If $\mathtt{f} = \mathtt{f}'$, then we know the predicate is true if and only if $s(l_2) = \mathcal{E}_s(G)$ and if $\mathtt{f} \neq \mathtt{f}'$ then the predicate propagates from the post- to the pre- condition. The following lemma captures the case where $\mathtt{f} \neq \mathtt{f}'$.

Lemma 2. *if* $s, h_1 \vdash \Sigma$, $s, h_2 \vdash x \mapsto \{\overline{\mathtt{f} : v}\}$, $s, h_2' \vdash x \mapsto \{\mathtt{f} : G, \ldots\}$, *and* $\mathtt{f} \neq \mathtt{f}'$ *then* $s, h_1 * h_2 \vdash x \overset{\mathtt{f}'}{\rightsquigarrow} l | \Sigma_1 * x \mapsto \{\overline{\mathtt{f} : v}\}$ *if and only if* $s, h_1 * h_2' \vdash x \overset{\mathtt{f}'}{\rightsquigarrow} l | \Sigma_1 * x \mapsto \{\mathtt{f} : G, \overline{\mathtt{f} : v}\}$.

Using the two Lemmas above, the following definition of $\mathcal{T}$ can be justified as properly fixing any embedded assertions $l_1 \overset{\mathtt{f}}{\rightsquigarrow} l_2$ in Π.

Definition 15. $\Pi' = \mathcal{T}(\Pi, \Sigma, \Sigma_A, \Sigma_B)$ *if and only if* $\Pi' = \mathcal{T}'(\Pi, \Sigma, \Sigma_A, \Sigma_B, \emptyset, \emptyset)$. *The two tail arguments here are members of* $Var \cup Var' \to \{\Sigma_1, \ldots, \Sigma_n\}$, *for some fixed* $\Sigma_1 * \ldots * \Sigma_n = \Sigma$. $\Pi' = \mathcal{T}'(\Pi, \Sigma, \Sigma_A, \Sigma_B, b, b')$ *is the function defined inductively on the structure of* Π *as follows.*

If Π is	*then Π' is*	*if the following condition holds:*
$v \in \Sigma_i \wedge \Pi_1$	$v \in \Sigma_i \wedge \Pi_1'$	$\Pi_1' = \mathcal{T}'(\Pi_1, \Sigma, \Sigma_A, \Sigma_B, b \cup (v \mapsto \Sigma_i), b' \cup (v \mapsto \Sigma_i))$
$\Pi_1 \wedge \Pi_2$	$\Pi_1' \wedge \Pi_2'$	$\Pi_1' = \mathcal{T}'(\Pi_1, \Sigma, \Sigma_A, \Sigma_B, b, b')$ *and*
$\Pi_1 \vee \Pi_2$	$\Pi_1' \vee \Pi_2'$	$\Pi_2' = \mathcal{T}'(\Pi_2, \Sigma, \Sigma_A, \Sigma_B, b, b')$
$\Pi = \neg\Pi_1$	$\Pi' = \neg\Pi_1'$	$\Pi_1' = \mathcal{T}'(\Pi_1, \Sigma, \Sigma_A, \Sigma_B, b, b')$
$E_1 = E_2$	$E_1 = E_2$	
$\forall v \in \Sigma_i.\Pi_1$	$\forall v \in \Sigma_i.\Pi_1'$	$\Pi_1' = \mathcal{T}'(\Pi_1, \Sigma, \Sigma_A, \Sigma_B, (v \mapsto \Sigma_i) \cup b / dom(b) -$
$\exists v.\Pi_1$	$\exists v.\Pi_1'$	$v, (v \mapsto \Sigma_i) \cup b' / dom(b') - v)$
$x_1 \overset{\mathtt{f}}{\rightsquigarrow} x_2$	$x_1 \overset{\mathtt{f}}{\rightsquigarrow} x_2$	$x_1 \in dom(b)$ *and* $b(x_1) = \Sigma_i$ *for some* $\Sigma_i \in \Sigma$
		$b(x_1) = \Sigma_A$ *and* $b'(x_1) = \Sigma_B$ *and* $\Sigma_A = x_1 \mapsto \{\overline{\mathtt{f} : v}, \mathtt{f}' : v'\}$ *and* $\Sigma_B = x_1 \mapsto \{\overline{\mathtt{f} : v}\}$ *and* $\mathtt{f} \neq \mathtt{f}'$
$x_1 \overset{\mathtt{f}}{\rightsquigarrow} x_2$	$x_2 = v'$	$\Sigma_A = x_1 \mapsto \{\overline{\mathtt{f} : v}, \mathtt{f}' : v'\}$ *and* $\Sigma_B = x_1 \mapsto \{\overline{\mathtt{f} : v}\}$ *and* $b(x_1) = \Sigma_A$ *and* $b'(x_1) = \Sigma_B$

The two auxiliary parameters in $\mathcal{T}'$ are mappings representing the bindings of variables in Π and Π' respectively. Also note that the two conditions for the second to last row in the table above are justified by Lemmas 1 and 2 above, respectively.

4.3 Unfold rule

The `as3` rule described above often requires the mutated heap component to be unfolded. This is accomplished using the `unfold` rule defined below.

$$\frac{\Pi_0' | \Sigma_0 * v_0 \mapsto (\overline{\mathtt{f} : v}) * \mathcal{R}_F(v_1, \bar{t}) * \ldots * \mathcal{R}_F(v_n, \bar{t})}{\Pi_0 | \Sigma_0 * \mathcal{R}_F(v_0, \bar{t})} (\mathtt{unfold})$$

where $\Pi_0' = unf(\Pi_0, \Sigma_0, R_F(t, \bar{t}), t \mapsto (\overline{\mathtt{f} : v}) * \mathcal{R}_F(v_1, \bar{t}) * \ldots * \mathcal{R}_F(v_n, \bar{t}))$. We define *unf* in Figure 4.4. The purpose of *unf* is to modify the Π portion of an abstract state to match the changes in Σ. The key issue in making these changes is to modify the $x \in \mathcal{R}_F(v_0, \bar{t})$, $\forall x \in \mathcal{R}_F(v_0, \bar{t}).body$ and $\exists x \in \mathcal{R}_F(v_0, \bar{t}).body$ which exist in Π. The $\mathcal{R}_F(v_0, \bar{t})$ needs to be replaced with its unfolded components and often subterms need to be duplicated.

4.4 Merging

The inference rule for if statements in Section 4.2 requires the pre-condition abstract states from their two branches to be identical. A $\Pi|\Sigma$ pair thus needs to be generated such that $\Pi|\Sigma$ implies $\Pi_1|\Sigma_1, \Pi_2|\Sigma_2$. Since the Π portion of the abstract state is simply first order logic, if $\Sigma_1 = \Sigma_2$, then we can generate $\Pi = \Pi_1 \wedge \Pi_2$.

The key issue that needs to be resolved is how to transform either Σ_1 or Σ_2 if they are not equal. In our example program the one rule we needed to introduce is a folding rule; other merge rules may also be needed in the future. The rule needed is as follows.

$$\frac{\Sigma * v \mapsto \{\overline{\mathtt{f} : x}\} * \mathcal{R}_S(x_1) * \ldots * \mathcal{R}_S(x_n) | \Pi}{\Sigma * \mathcal{R}_S(v) | \Pi'} (\mathtt{fold})$$

$\Pi' = \mathit{unf}(\Pi, \Sigma_u, \Sigma_t, \Sigma_e)$ is defined inductively as follows:

This holds	if the following holds
$\mathit{unf}(\Pi_0 \wedge \Pi_1, \Sigma_u, \Sigma_t, \Sigma_e) = \Pi_0' \wedge \Pi_1'$	$\Pi_0' = \mathit{unf}(\Pi_0, \Sigma_u, \Sigma_t, \Sigma_e)$ and $\Pi_1' = \mathit{unf}(\Pi_1, \Sigma_u, \Sigma_t, \Sigma_e)$
$\mathit{unf}(\Pi_0 \vee \Pi_1, \Sigma_u, \Sigma_t, \Sigma_e) = \Pi_0' \vee \Pi_1'$	
$\mathit{unf}(\neg\Pi, \Sigma_u, \Sigma_t, \Sigma_e) = \neg\Pi'$	$\Pi' = \mathit{unf}(\Pi, \Sigma_u, \Sigma_t, \Sigma_e)$
$\mathit{unf}(\forall v \in \Sigma_t'.\Pi, \Sigma_u, \Sigma_t, \Sigma_e) = \forall v \in \Sigma_t'.\Pi'$	$\Pi' = \mathit{unf}(\Pi, \Sigma_u, \Sigma_t, \Sigma_e)$ and $\Sigma_t' \neq \Sigma_t$
$\mathit{unf}(\exists v \in \Sigma_t'.\Pi, \Sigma_u, \Sigma_t, \Sigma_e) = \exists v \in \Sigma_t'.\Pi'$	
$\mathit{unf}(l_1 \overset{\mathtt{f}}{\rightsquigarrow} l_2, \Sigma_u, \Sigma_t, \Sigma_e) = l_1 \overset{\mathtt{f}}{\rightsquigarrow} l_2$	
$\mathit{unf}(E_1 = E_2, \Sigma_u, \Sigma_t, \Sigma_e) = E_1 = E_2$	
$\mathit{unf}(v \in \Sigma_i \wedge \Pi, \Sigma_u, \Sigma_t, \Sigma_e) = v \in \Sigma_i \wedge \Pi'$	$\Sigma = \ldots * \Sigma_i * \ldots$ and $\Pi' = \mathit{unf}(\Pi, \Sigma_u, \Sigma_t, \Sigma_e)$
$\mathit{unf}(x' \in \Sigma_t \wedge \Pi, \Sigma_u, \Sigma_t, \Sigma_e) =$ $x \in (v_0 \mapsto \{\overline{f : v}\} * R_F(v_1, \overline{t}) * \ldots * R_F(v_n, \overline{t})) \wedge \Pi'$	$\Pi' = \mathit{unf}(\Pi, \Sigma_u, \Sigma_t, \Sigma_e)$
$\mathit{unf}(x \in \Sigma_t \wedge \Pi, \Sigma_u, \Sigma_t, \Sigma_e) =$ $x_0 \in v_0 \mapsto \{\overline{f : v}\} \wedge$ $x_1 \in R_F(v_1, \overline{t}) \wedge \ldots \wedge$ $x_n \in R_F(v_n, \overline{t}) \wedge$ $(\Pi'[x_0/x] \wedge \ldots \wedge \Pi'[x_n/x])$	x is not a primed variable. $\Pi' = \mathit{unf}(\Pi, \Sigma_u, \Sigma_t, \Sigma_e)$ $x_0, \ldots, x_n$ are fresh $\Sigma_e = v_0 \mapsto \{\overline{f : v}\} * R_F(v_1, \overline{t}) * \ldots * R_F(v_n, \overline{t})$
$\mathit{unf}(\forall x \in \Sigma_t.\Pi, \Sigma_u, \Sigma_t, \Sigma_e) =$ $\Pi'[x_0/x] \wedge$ $\forall x \in R_F(v_1, \overline{t}).\Pi' \wedge$ $\ldots$ $\forall x \in R_F(v_n, \overline{t}).\Pi'$	
$\mathit{unf}(\exists x \in \Sigma_t.\Pi, \Sigma_u, \Sigma_t, \Sigma_e) =$ $\Pi'[x_0/x] \vee$ $\exists x \in R_F(v_1, \overline{t}).\Pi' \vee$ $\ldots$ $\exists x \in R_F(v_n, \overline{t}).\Pi'$	

Fig. 3. The definition of *unf*

where $\overline{f} = S$ and $\Pi' = \mathit{foldPi}(\Pi, \Pi, \mathcal{R}_S(v), \{\mathcal{R}_S(x_1), \ldots, \mathcal{R}_S(x_n)\})$. The auxiliary *foldPi* function is now given.

Definition 16. *The function* $\Pi' = \mathit{foldPi}(\Pi, \Pi^r, \mathcal{R}_S(v), \overline{\Sigma})$ *is defined by the table below:*

If Π *is*	*then* Π'*is*	*if the following condition holds:*
$\Pi_1 \wedge \Pi_2$	$\Pi_1' \wedge \Pi_2'$	$\Pi_1' = \mathit{foldPi}(\Pi_1, \Pi^r, \mathcal{R}_S(v), \overline{\Sigma})$ *and* $\Pi_2' = \mathit{foldPi}(\Pi_2, \Pi^r, \mathcal{R}_S(v), \overline{\Sigma})$
$\Pi_1 \vee \Pi_2$	$\Pi_1' \vee \Pi_2'$	
$\neg\Pi_1$	$\neg\Pi_1'$	$\Pi_1' = \mathit{foldPi}(\Pi_1, \Pi^r, \mathcal{R}_S(v), \overline{\Sigma})$
$E_1 = E_2$	$E_1 = E_2$	
$v \in \Sigma' \wedge \Pi_1$	$v \in \Sigma' \wedge \Pi_1'$	$\Sigma' \notin \overline{\Sigma}$
$\forall v \in \Sigma'.\Pi_1$	$\forall v \in \Sigma'.\Pi_1'$	$\Sigma' \notin \overline{\Sigma}$
$\exists v \in \Sigma'.\Pi_1$	$\exists v \in \Sigma'.\Pi_1'$	
$\forall v \in \Sigma'.\Pi_1$	$\forall v \in \mathcal{R}_S(v).\Pi_1'$	$\Sigma' \in \overline{\Sigma}$ *and* $\Pi^r \rightarrow \forall v \in \mathcal{R}_S(v).\Pi_1'$ *is valid*
$\exists v \in \Sigma'.\Pi_1$	$\exists v \in \mathcal{R}_S(v).\Pi_1'$	$\Sigma' \in \overline{\Sigma}$ *and* $\Pi^r \rightarrow \exists v \in \mathcal{R}_S(v).\Pi_1'$ *is valid*
$x_1 \overset{\mathtt{f}}{\rightsquigarrow} x_2$	$x_1 \overset{\mathtt{f}}{\rightsquigarrow} x_2$	

4.5 Simplification and strengthening rules

Beyond the above rules there are a large number of rules needed for more basic reasoning, too many to put in this paper (but of course all are in the Coq development). We for example have the following axioms to simplify spatial heap descriptions.

$$\Sigma * \texttt{emp} = \Sigma \qquad\qquad \mathcal{R}_R(\texttt{nil}, \overline{\texttt{f}}) = \texttt{emp}$$

For the Π portion, there are all the usual propositional logic rules (for example, $a \wedge \texttt{true} \rightarrow a$). Axioms for $\rightsquigarrow$ and $\exists$ for example include the following:

$$\neg(\texttt{nil} \overset{\texttt{f}}{\rightsquigarrow} x) \qquad\qquad E \in \Sigma \rightarrow \exists x \in \Sigma . x = E$$

5 Outline of Coq proof of the example

To give a feel for how the correctness proof for the example in Section 2 was performed in Coq, we highlight a few portions of the proof here. We start by outlining the first few steps in which Hoare axioms are used to decompose the proof obligation for the entire program into obligations that implement back chaining on the individual statements. We then show how an unfold is done to allow an `as3` axiom to be applied.

5.1 Using Hoare axioms to back chain

We walk through the first few Coq steps in the the correctness proof. We start with a single proof obligation for the following Hoare triple:

$$\begin{array}{l}\{\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p) | \\ \quad\quad \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m \wedge \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m\} \\ \quad \texttt{while(t)}\{\ldots\} \\ \{\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p) | t = nil \wedge \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m\}\end{array}$$

We first apply our while loop axiom `absWhile` with the invariant:

$$\begin{array}{l}\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p) | \\ \quad \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m \wedge \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m\end{array}$$

This generates three proof obligations corresponding to the three preconditions of the rule. The first is the most interesting–it is the obligation to prove the correctness of the body:

$$\begin{array}{l}\{\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p) | \\ \quad\quad \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m \wedge \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m \wedge t \neq nil\} \\ \quad \texttt{n} = \texttt{p}; \\ \quad \ldots \\ \quad \texttt{if(t} \rightarrow \texttt{l} == \texttt{NULL} \;\&\&\; \texttt{t} \rightarrow \texttt{r} == \texttt{NULL)}\{\ldots\}\texttt{else}\{\ldots\} \\ \{\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p) | \\ \quad\quad \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m \wedge \forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m\}\end{array}$$

The other two obligations verify that the pre-condition implies the invariant and that the invariant implies the post condition.

We then use the `absSeq` rule to break up the sequence of statements into individual proof obligations. We use Coq's `eapply` tactic which leaves the pre-conditions existentially quantified. We first get an obligation for the if block which appears as follows:

$$\begin{array}{l}\{?1|?2\}\\ \quad \texttt{if}(\texttt{t} \rightarrow \texttt{l} == \texttt{NULL} \;\&\&\; \texttt{t} \rightarrow \texttt{r} == \texttt{NULL})\{\ldots\}\texttt{else}\{\ldots\}\\ \{\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p)|\\ \qquad \forall n \in \mathcal{R}_n(i).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m \wedge \forall n \in \mathcal{R}_n(i).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m\}\end{array}$$

These existential obligations `?1`/`?2` get filled in as we back trace individual statements. As an example, one of the obligations we will eventually work down to is an obligation for an assignment that looks like this:

$$\begin{array}{l}\{?1|?2\}\\ \quad \texttt{t} = \texttt{t} \rightarrow \texttt{l}\\ \{\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p)|\\ \qquad \forall n \in \mathcal{R}_n(i).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m \wedge \forall n \in \mathcal{R}_n(i).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m\}\end{array}$$

Here we apply `as2`, and then apply a number of of auxiliary axioms. This produces the following pre-condition.

$$\begin{array}{l}\{\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p)|\\ \qquad t \overset{l}{\rightsquigarrow} v' \wedge \forall n \in \mathcal{R}_n(i).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m \wedge \forall n \in \mathcal{R}_n(i).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m\}\end{array}$$

which replaces the existential variables.

5.2 Unfolding prior to applying `as3`

Often we need to do an unfolding prior to performing a back chaining operation. Consider the following proof obligation:

$$\begin{array}{l}\{?1|?2\}\\ \quad \texttt{i} \rightarrow \texttt{t} = \texttt{x};\\ \{\mathcal{R}_{lr}(r) * \mathcal{R}_n(i) * \mathcal{R}_n(p)|\\ \qquad t \in \mathcal{R}_{lr}(r) \wedge t \overset{l}{\rightsquigarrow} v' \wedge \forall n \in \mathcal{R}_n(i).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m \wedge\\ \qquad \forall n \in \mathcal{R}_n(p).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m\}\end{array}$$

The problem here is that the assignment requires us to reason about an individual cell within the $\mathcal{R}_n(i)$ construct. Ideally, we would like to use the `as3` rule to generate the precondition. However, the rule will not match with the abstract state in its current form. We need to apply our `unfold` rule from section 4.3. This yields the following:

$$\begin{array}{l}\{?1|?2\}\\ \quad \texttt{i} \rightarrow \texttt{t} = \texttt{x};\\ \{i \mapsto \{n : n, ?3\} * \mathcal{R}_{lr}(r) * \mathcal{R}_n(n) * \mathcal{R}_n(p)|\\ \qquad t \in \mathcal{R}_{lr}(r) \wedge t \overset{l}{\rightsquigarrow} v' \wedge \forall n \in \mathcal{R}_n(p).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m \wedge\\ \qquad (\forall n \in \mathcal{R}_n(i).\exists m \in \mathcal{R}_{lr}(r).n \overset{p}{\rightsquigarrow} m) \wedge \exists m \in \mathcal{R}_{lr}(r) i \overset{p}{\rightsquigarrow} m\}\end{array}$$

Now we can apply our `as3` rule to fill in the precondition yielding the following:

$$\begin{array}{l}\{i \mapsto \{n : n\} * \mathcal{R}_{lr}(r) * \mathcal{R}_n(n) * \mathcal{R}_n(p) | \\ \quad t \in \mathcal{R}_{lr}(r) \wedge t \overset{l}{\rightsquigarrow} v' \wedge \forall n \in \mathcal{R}_n(p). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m \wedge \\ \quad (\forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m) \wedge \exists m \in \mathcal{R}_{lr}(r) i \overset{p}{\rightsquigarrow} m\} \\ \texttt{i} \rightarrow \texttt{t} = \texttt{x}; \\ \{i \mapsto \{n : n, t : x\} * \mathcal{R}_{lr}(r) * \mathcal{R}_n(n) * \mathcal{R}_n(p) | \\ \quad t \in \mathcal{R}_{lr}(r) \wedge t \overset{l}{\rightsquigarrow} v' \wedge \forall n \in \mathcal{R}_n(p). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m \wedge \\ \quad (\forall n \in \mathcal{R}_n(i). \exists m \in \mathcal{R}_{lr}(r). n \overset{p}{\rightsquigarrow} m) \wedge \exists m \in \mathcal{R}_{lr}(r) i \overset{p}{\rightsquigarrow} m\}\end{array}$$

We have shown a few illustrative steps in this section to give a flavor of how Coq verifications using our framework are constructed.

6 Conclusions

We have constructed extensions to traditional separation logic that allow it to reason about more complex data structures than previous work. The example tree traversal program we presented contains data structures with fairly complex pointer relationships. Notably, the traversal is represented as a linked list of pointers into a tree; existing separation logics cannot handle such a relationship. The invariant proof of this program that was formalized in Coq guarantees that the program maintains the integrity of this data structure and there are no errors such as memory leaks or dangling pointer references. These more complex data structures present a much greater verification challenge, and here we have shown the feasibility of using Coq to fully verify programs with these complex data structure invariants. To accomplish our goals we needed to develop novel axioms to fold and unfold our new recursive separation logic constructs which are in a form amenable to formal verification.

Currently there are many manual steps needed to complete the proof. We anticipate that much of this can be automated through the development of tactics, and that is a subject of future work. There already has been progress in developing tactics for separation logic proofs in Coq [12, 1] that we can build on. Our long-term goal is to get to the point where the only user input required will be the pre- and post- conditions in the form of abstract states as well as loop invariants. We anticipate that programs will be annotated with these conditions in a manner similar to [4]. Because the language for the abstraction is much richer than many of the previous works, automatic abstraction from a concrete domain such as [9] is likely to be difficult.

Another future goal is to formally verify our new axioms in Coq, following the program of [2] where the core axioms of separation logic are verified in Coq. It should be pointed out that there are many related projects pursuing formal theorem proving of imperative program properties in separation logic, including [11, 17] which are more ambitious in terms of the examples verified; our contribution here is to extend the potential for future verifications to more complex heap shapes.

Acknowledgments The authors would like to thank Mike Hicks, Henny Sipma and Steven Magill for reviewing drafts of this paper. The first author would also like to thank Dino Distefano, Honseok Yang and other members of the East London Massive for early feedback and pointing out key research.

References

1. A. W. Appel. Tactics for separation logic, 2006. Early draft.
2. Andrew Appel and Sandrine Blazy. Separation logic for small-step cminor. In *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. 2007.
3. Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*.
4. Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers.
5. Peter W. O'Hearn Dino Distefano and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
6. S. P. Rahul Westley Weimer George C. Necula, Scott McPeak. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction*, March 2002.
7. Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
8. B Cook D Distefano PW O'Hearn T Wies J Berdine, C Calcagno and H Yang. Shape analysis for composite data structures. In *CAV'07*. Springer, 2007.
9. Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. Inferring invariants in separation logic for imperative list-processing programs. In *SPACE*, 2006.
10. Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation logic. *Information and Media Technologies*, 4(2):304–316, 2009.
11. Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. 2006.
12. Andrew Mccreight. Practical tactics for separation logic. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 343–358, 2009.
13. Magnus Myreen. Separation logic adapted for proofs by rewriting. In *Interactive Theorem Proving Conference*. Springer, 2010.
14. Hongseok Yang Peter O'Hearn, John Reynolds. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume LNCS 2142, pages 1–19, 2001.
15. B. Cook G. Ramalingam M. Sagiv R. Manevich, J. Berdine. Shape analysis by graph decomposition.
16. Reinhard Wilhel Shmuel Sagiv, Thomas W. Reps. Solving shape-analysis problems in languages with destructive updating. 20(1):1–50, 1998.
17. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 97–108, 2007.