Symbolic Execution with Separation Logic

Josh Berdine[1], Cristiano Calcagno[2], and Peter W. O'Hearn[1]

[1] Queen Mary, University of London
[2] Imperial College, London

Abstract. We describe a sound method for automatically proving Hoare triples for loop-free code in Separation Logic, for certain preconditions and postconditions (symbolic heaps). The method uses a form of symbolic execution, a decidable proof theory for symbolic heaps, and extraction of frame axioms from incomplete proofs. This is a precursor to the use of the logic in automatic specification checking, program analysis, and model checking.

1 Introduction

Separation Logic has provided an approach to reasoning about programs with pointers that often leads to simpler specifications and program proofs than previous formalisms [12]. This paper is part of a project attempting to transfer the simplicity of the by-hand proofs to a fully automatic setting.

We describe a method for proving Hoare triples for loop-free code, by a form of symbolic execution, for certain (restricted) preconditions and postconditions. It is not our intention here to try to show that the method is useful, just to say what it is, and establish its soundness. This is a necessary precursor to further possible developments on using Separation Logic in:

- *Automatic Specification Checking*, where one takes an annotated program (with preconditions, postconditions and loop invariants) and chops it into triples for loop-free code in the usual way;
- *Program Analysis*, where one uses fixed-point calculations to remove or reduce the need for annotations; and
- *Software Model Checking*.

The algorithms described here are, in fact, part an experimental tool of the first variety, Smallfoot. Smallfoot itself is described separately in a companion paper [2]; here we confine ourselves to the technical problems lying at its core. Of course, program analysis and model checking raise further problems – especially, the structure of our "abstract" domain and the right choice of widening operators [3] – and further work is under way on these.

There are three main issues that we consider.

1. How to construe application of Separation Logic proof rules as symbolic execution. The basic idea can be seen in the axiom

$$\{A * x{\mapsto}[f\colon y]\}\ x{\to}f{:=}z\ \{A * x{\mapsto}[f\colon z]\}$$

where the precondition is updated in-place, in a way that mirrors the imperative update of the actual heap that occurs during program execution. The separating

K. Yi (Ed.): APLAS 2005, LNCS 3780, pp. 52–68, 2005.

conjunction, $*$, short-circuits the need for a global alias check in this axiom. $A * x{\mapsto}[f\colon y]$ says that the heap can be partitioned into a single cell x, that points to (has contents) a record with y in its f field, and the rest of the heap, where A holds. We know that A will continue to hold in the rest of the heap if we update x, because x is not in A's part of the heap.

There are two restrictions on assertions which make the symbolic execution view tenable. First, we restrict to a format of the form $B \wedge S$ where B is a pure boolean formula and S is a $*$-combination of heap predicates. We think of these assertions as "symbolic heaps"; the format makes the analogy with the in-place aspect of concrete heaps apparent. Second, the preconditions and postconditions do not describe the detailed contents of data structures, but rather describe shapes (in roughly the sense of the term used in shape analysis). Beyond the basic primitives of Separation Logic, Smallfoot at this point includes several hardwired shape predicates for: singly- and doubly-linked lists, xor-linked lists, and trees. Here we describe our results for singly-linked lists and trees only.

2. How to discharge entailments $A \vdash B$ between symbolic heaps. We give a decidable proof theory for the assertions in our language.

One key issue is how to account for entailments that would normally require induction. To see the issue, consider a program for appending two lists. When you get to the end of the first list you link it up to the second. At this point to prove the program requires showing an entailment

$$\mathsf{ls}(x,t) * t{\mapsto}[n\colon y] * \mathsf{ls}(y,\mathsf{nil}) \vdash \mathsf{ls}(x,\mathsf{nil})$$

where we have a list segment from x to t, a single node t, and a further segment (the second list) from y up to nil. The entailment itself does not follow at once from simple unwinding of an inductive definition of list segments. In the metatheory it is proven by induction, and in our proof theory it will be handled using rules that are consequences of induction but that are themselves non-inductive in character.

In [1] we showed decidability of a fragment of the assertion language of this paper, concentrating on list segments. Here we give a new proof procedure, which appears to be less fragile in the face of extension than the model-theoretic procedure of [1], since if the fragment is extended with additional formulæ, then the decision procedure of [1] remains complete but potentially becomes unsound, while the present proof theory remains sound but potentially becomes incomplete. Additionally, and crucially, it supports inference of frame axioms.

3. Inference of Frame Axioms. Separation Logic allows specifications to be kept small because it avoids the need to state frame axioms, which describe the portions of the heap not altered by a command [10]. To see the issue, consider a specification

$$\{\mathsf{tree}(p)\}\ \mathrm{disp\_tree}(p)\ \{\mathsf{emp}\}$$

for disposing a tree, which just says that if you have a tree (and nothing else) and you dispose it, then there is nothing left. When verifying a recursive procedure for disposing a tree there will be recursive calls for disposing subtrees. The problem is that, generally, a precondition at a call site will not match that for the procedure due to extra heap around. For example, at the site of a call

disp_tree(i) to dispose the left subtree we might have a root pointer p and the right subtree j as well as the left subtree – $p{\mapsto}[l{:}\, i, r{:}\, j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}$ – while the precondition for the overall procedure specification expects only a single tree.

Separation Logic has a proof rule, the Frame Rule, which allows us to resolve this mismatch. It allows us the first step in the inference:

$$\dfrac{\dfrac{\{\mathsf{tree}(i)\}\ \text{disp\_tree}(i)\ \{\mathsf{emp}\}}{\{p{\mapsto}[l{:}\, i, r{:}\, j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}\ \text{disp\_tree}(i)\ \{p{\mapsto}[l{:}\, i, r{:}\, j] * \mathsf{emp} * \mathsf{tree}(j)\}}}{\{p{\mapsto}[l{:}\, i, r{:}\, j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}\ \text{disp\_tree}(i)\ \{p{\mapsto}[l{:}\, i, r{:}\, j] * \mathsf{tree}(j)\}}$$

To automatically generate proof steps like this we need some way to infer frame axioms, the leftover parts (in this case $p{\mapsto}[l{:}\, i, r{:}\, j] * \mathsf{tree}(j)$). Sometimes, this leftover part can be found by simple pattern matching, but often not. In this paper we describe a novel method of extracting frame axioms from incomplete proofs in our proof theory for entailments. A failed proof can identify the "left-over" part which, were you to add it in, would complete the proof, and we show how this can furnish us with a sound choice of frame axiom.

The notion of symbolic execution presented in this paper is, in a general sense, similar in spirit to what one obtains in Shape Analysis or PALE [14, 7]. However, there are nontrivial differences in the specifics. In particular, we have been unsuccessful in attempts to compositionally translate Separation Logic into either PALE's assertion language or into a shape analysis; the difficulty lies in treating the separating conjunction connective. And this is the key to employing the frame rule, which is responsible for Separation Logic's small specifications of procedures. So it seems sensible to attempt to describe symbolic execution for Separation Logic directly, in its own terms.

2 Symbolic Heaps

The concrete heap models assume a fixed finite collection Fields, and disjoint sets Loc of locations, Val of non-addressable values, with $\mathsf{nil} \in \text{Val}$. We then set:

$$\begin{aligned} \text{Heaps} &\stackrel{\text{def}}{=} \text{Loc} \stackrel{\text{fin}}{\rightharpoonup} (\text{Fields} \to \text{Val} \cup \text{Loc}) \\ \text{Stacks} &\stackrel{\text{def}}{=} \text{Var} \to \text{Val} \cup \text{Loc} \end{aligned}$$

As a language for reasoning about these models we consider certain pure (heap independent) and spatial (heap dependent) assertions.

$$\begin{array}{rll} x, y, \ldots \in \text{Var} & & \text{variables} \\ E ::= \mathsf{nil} \mid x & & \text{expressions} \\ P ::= E{=}E \mid \neg P & & \text{simple pure formulæ} \\ \Pi ::= \mathsf{true} \mid P \mid \Pi \wedge \Pi & & \text{pure formulæ} \\ f, f_i, \ldots \in \text{Fields} & & \text{fields} \\ \rho ::= f_1{:}\, E_1, \ldots, f_k{:}\, E_k & & \text{record expressions} \\ S ::= E{\mapsto}[\rho] & & \text{simple spatial formulæ} \end{array}$$

$$\begin{array}{rl} \Sigma ::= \mathsf{emp} \mid S \mid \Sigma * \Sigma & \text{spatial formulæ} \\ \Pi \mathbin{!} \Sigma & \text{symbolic heaps} \end{array}$$

Symbolic heaps are pairs $\Pi \mathbin{!} \Sigma$ where Π is essentially an $\wedge$-separated sequence of pure formulæ, and Σ a $*$-separated sequence of simple spatial formulæ.[1] The pure part here is oriented to stating facts about pointer programs, where we will use equality with nil to indicate a situation where we do not have a pointer. Other subsets of boolean logic could be considered in other situations.

In this heap model a location maps to a record of values. The formula $E{\mapsto}[\rho]$ can mention any number of fields in ρ, and the values of the remaining fields are implicitly existentially quantified. This allows us to write specifications which do not mention fields whose values we do not care about.

The semantics is given by a forcing relation $s, h \vDash A$ where $s \in \text{Stacks}$, $h \in \text{Heaps}$, and A is a pure assertion, spatial assertion, or symbolic heap. $h = h_0 * h_1$ indicates that the domains of h_0 and h_1 are disjoint, and h is their graph union.

$$[\![x]\!]s \stackrel{\text{def}}{=} s(x) \qquad\qquad [\![\mathsf{nil}]\!]s \stackrel{\text{def}}{=} \text{nil}$$

$$\begin{array}{lll} s, h \vDash E_1{=}E_2 & \stackrel{\text{def}}{\text{iff}} & [\![E_1]\!]s = [\![E_2]\!]s \\ s, h \vDash \neg P & \stackrel{\text{def}}{\text{iff}} & s, h \nvDash P \\ s, h \vDash \mathsf{true} & & \text{always} \\ s, h \vDash \Pi_0 \wedge \Pi_1 & \stackrel{\text{def}}{\text{iff}} & s, h \vDash \Pi_0 \text{ and } s, h \vDash \Pi_1 \\ s, h \vDash E_0{\mapsto}[f_1{:}\,E_1, \ldots, f_k{:}\,E_k] & \stackrel{\text{def}}{\text{iff}} & h = [[\![E_0]\!]s{\to}r] \text{ where } r(f_i) = [\![E_i]\!]s \text{ for } i \in 1..k \\ s, h \vDash \mathsf{emp} & \stackrel{\text{def}}{\text{iff}} & h = \varnothing \\ s, h \vDash \Sigma_0 * \Sigma_1 & \stackrel{\text{def}}{\text{iff}} & \exists h_0 h_1 .\, h = h_0{*}h_1 \text{ and } s, h_0 \vDash \Sigma_0 \text{ and } s, h_1 \vDash \Sigma_1 \\ s, h \vDash \Pi \mathbin{!} \Sigma & \stackrel{\text{def}}{\text{iff}} & s, h \vDash \Pi \text{ and } s, h \vDash \Sigma \end{array}$$

To reason about pointer programs one typically needs predicates that describe inductive properties of the heap. We describe two of the predicates (adding to the simple spatial formulæ) that we have experimented with in Smallfoot.

2.1 Trees

We describe a model of binary trees where each internal node has fields l, r for the left and right subtrees. The empty tree is given by nil. What we require is that $\mathsf{tree}(E)$ is the least (logically strongest) predicate satisfying:

$$\mathsf{tree}(E) \iff (E = \mathsf{nil} \wedge \mathsf{emp}) \vee (\exists x, y.\, E{\mapsto}[l{:}\,x, r{:}\,y] * \mathsf{tree}(x) * \mathsf{tree}(y))$$

where x and y are fresh. The use of the $*$ between $E{\mapsto}[l{:}\,x, r{:}\,y]$ and the two subtrees ensures that there are no cycles, and the $*$ between the subtrees ensures that there is no sharing; it is not a DAG.

[1] Note that we abbreviate $\neg(E_1{=}E_2)$ as $E_1{\neq}E_2$ and $\mathsf{true} \mathbin{!} \Sigma$ as Σ, and use $\equiv$ to denote "syntactic" equality of formulæ, which are considered up to symmetry of $=$, permutations across $\wedge$ and $*$, *e.g.*, $\Pi \wedge P \wedge P' \equiv \Pi \wedge P' \wedge P$, involutivity of negation, and unit laws for true and emp. We use notation treating formulæ as sets of simple formulæ, *e.g.*, writing $P \in \Pi$ for $\Pi \equiv P \wedge \Pi'$ for some Π'.

The way that the record notation works allows this definition to apply to any heap model that contains at least l and r fields. In case there are further fields, say a field d for the data component of a node, the definition is independent of what the specific values are in those fields.

Our description of this predicate is not entirely formal, because we do not have existential quantification, disjunction, or recursive definitions in our fragment. However, what we are doing is defining a new simple spatial formula (extending syntactic category S above), and we are free to do that in the metatheory. A longer-winded way to view this, as a semantic definition, is to say that it is the least predicate such that

$s, h \vDash \mathsf{tree}(E)$ holds if and only if

1. $s, h \vDash E = \mathsf{nil} \wedge \mathsf{emp}$, or
2. ℓ_x, ℓ_y exist where $(s \mid x{\rightarrow}\ell_x, y{\rightarrow}\ell_y), h \vDash E{\mapsto}[l\colon x, r\colon y] * \mathsf{tree}(x) * \mathsf{tree}(y)$

Of course, we would have to prove (in the metatheory) that the least definition exists, but that is not difficult.

2.2 List Segments

We will work with linked lists that use field n for the next element. The predicate for linked list segments is the least satisfying the following specification:

$$\mathsf{ls}(E, F) \iff (E{=}F \wedge \mathsf{emp}) \vee (E{\neq}F \wedge \exists y. E{\mapsto}[n\colon y] * \mathsf{ls}(y, F))$$

Once again, this definition allows for additional fields, such as a head field, but the ls predicate is insensitive to the values of these other fields.

With this definition a complete linked list is one that satisfies $\mathsf{ls}(E, \mathsf{nil})$. Complete linked lists, or trees for that matter, are much simpler than segments. But the segments are sometimes needed when reasoning in the middle of a list, particularly for iterative programs. (Similar remarks would apply to iterative tree programs.)

2.3 Examples

For some context and a feel for the sorts of properties expressible, we present a few example procedures with specifications in the fragment in Table 1. We do not discuss loops and procedures in the technical part of the paper, but the techniques we present are strong enough to verify these programs and are used in Smalllfoot to do so.

The disp_tree(p) example accepts any heap in which the argument points to a tree and deallocates the tree, returning the empty heap. As discussed earlier, proving this requires inferring frame axioms at the recursive call sites. Also, this example demonstrates the ability to specify absence of memory leaks, since, if the `dispose p;` command was omitted, then the specification would not hold.

While $*$ is required in the proof of disp_tree, it does not appear in the specification. The second example illustrates the use of $*$ in specifications, where copy_tree guarantees that after copying a tree, the input tree and the new copy occupy disjoint memory cells.

The third example is the source of the entailment requiring induction discussed in the introduction. This procedure also illustrates how list segments are

Table 1. Example Programs

```
disp_tree(;p)
[tree(p)] {
  local i,j;
  if (p==nil) {}
  else {
    i = p->l;
    j = p->r;
    disp_tree(;i);
    disp_tree(;j);
    dispose p; }
} [emp]
```

```
copy_tree(q;p)
[tree(p)] {
  local i,j,i',j';
  if (p==nil) {}
  else {
    i = p->l;
    j = p->r;
    copy_tree(i';i);
    copy_tree(j';j);
    q = cons();
    q->l = i';
    q->r = j'; }
} [tree(q) ∗ tree(p)]
```

```
append_list(x;y)
[ls(x, nil) ∗ ls(y, nil)] {
  local t,u;
  if (x==nil) {
    x = y; }
  else {
    t = x; u = t->n;
    while (u!=nil)
    [ls(x, t) ∗ t↦[n: u] ∗ ls(u, nil)]
    { t = u;
      u = t->n; }
    t->n = y; }
} [ls(x, nil)]
```

Note that in these examples, assertions are enclosed in square brackets, and procedure parameter lists consist first of the reference parameters, followed by a semicolon, and finally the value parameters.

sometimes needed in loop invariants of code whose specifications only involve complete lists ending in nil.

3 Symbolic Execution

In this section we give rules for triples of the form

$$\{\Pi \mid \Sigma\}\ C\ \{\Pi' \mid \Sigma'\}$$

where C is a loop-free program. The commands C are given by the grammar:

$$C ::= \texttt{empty} \mid x{:=}\,E\ ;\ C \mid x{:=}\,E{\to}f\ ;\ C \mid E{\to}f{:=}\,F\ ;\ C$$
$$\mid\ \texttt{new}(x)\ ;\ C \mid \texttt{dispose}(E)\ ;\ C \mid \texttt{if}\ P\ \texttt{then}\ C\ \texttt{else}\ C\ \texttt{fi}\ ;\ C$$

The rules in this section appeal to entailments $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$ between symbolic heaps. Semantically, entailment is defined by:

$$\Pi \mid \Sigma \vdash \Pi' \mid \Sigma' \text{ is true} \quad \text{iff} \quad \forall s, h.\ s, h \vDash \Pi \mid \Sigma \text{ implies } s, h \vDash \Pi' \mid \Sigma'$$

For the presentation of rules in this section we will regard semantic entailment as an oracle. Soundness of symbolic execution just requires an approximation.

3.1 Operational Rules

The operational rules use the following notation for record expressions:

$$\mathit{mutate}(\rho, f, F) = \begin{cases} f{:}\,F, \rho' & \text{if } \rho = f{:}\,E, \rho' \\ f{:}\,F, \rho & \text{if } f \notin \rho \end{cases} \qquad \mathit{lookup}(\rho, f) = \begin{cases} E & \text{if } \rho = f{:}\,E, \rho' \\ x \text{ fresh} & \text{if } f \notin \rho \end{cases}$$

The fresh variable returned in the lookup case corresponds to the idea that if a record expression does not give a value for a particular field then we do not

Table 2. Operational Symbolic Execution Rules

$$\textsc{Empty}\ \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\{\Pi \mid \Sigma\}\ \texttt{empty}\ \{\Pi' \mid \Sigma'\}} \qquad \textsc{Assign}\ \frac{\{x{=}E[x'/x] \wedge (\Pi \mid \Sigma)[x'/x]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ x{:=}E\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}$$

$$\textsc{Lookup}\ \frac{\{x{=}F[x'/x] \wedge (\Pi \mid \Sigma * E{\mapsto}[\rho])[x'/x]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ x{:=}E{\to}f\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}, \mathit{lookup}(\rho, f) = F$$

$$\textsc{Mutate}\ \frac{\{\Pi \mid \Sigma * E{\mapsto}[\rho']\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ E{\to}f{:=}F\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ \mathit{mutate}(\rho, f, F) = \rho'$$

$$\textsc{New}\ \frac{\{(\Pi \mid \Sigma)[x'/x] * x{\mapsto}[]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ \texttt{new}(x)\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}$$

$$\textsc{Dispose}\ \frac{\{\Pi \mid \Sigma\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ \texttt{dispose}(E)\ ;\ C\ \{\Pi' \mid \Sigma'\}}$$

$$\textsc{Conditional}\ \frac{\{\Pi \wedge P \mid \Sigma\}\ C_1\ ;C\ \{\Pi' \mid \Sigma'\} \qquad \{\Pi \wedge \neg P \mid \Sigma\}\ C_2\ ;C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ \texttt{if}\ P\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi}\ ;\ C\ \{\Pi' \mid \Sigma'\}}$$

care what it is. These definitions do not result in conditionals being inserted into record expressions; they do not depend on the values of variables or the heap.

The operational rules are shown in Table 2. One way to understand these rules is by appeal to operational intuition. For instance, reading bottom-up, from conclusion to premise, the MUTATE rule says: To determine if $\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ E{\to}f{:=}F\ ;\ C\ \{\Pi' \mid \Sigma'\}$ holds, execute $E{\to}f{:=}F$ on the symbolic pre-state $\Pi \mid \Sigma * E{\mapsto}[\rho]$, updating E in place, and then continue with C. Likewise, the DISPOSE rule says to dispose a symbolic cell (a $\mapsto$ fact), the NEW rule says to allocate, and the LOOKUP rule to read. The substitutions of fresh variables are used to keep track of (facts about) previous values of variables.

The role of fresh variables can be understood in terms of standard considerations on Floyd-Hoare logic. Recall that in Floyd's assignment axiom

$$\{A\}\ x{:=}E\ \{\exists x'.\ x{=}E[x'/x] \wedge A[x'/x]\}$$

the fresh variable x' is used to record (at least the existence of) a previous value for x. Our fragment here is quantifier-free, but we can still use the same general idea as in the Floyd axiom, as long as we have an overall postcondition and a continuation of the assignment command.

$$\frac{\{x{=}E[x'/x] \wedge A[x'/x]\}\ C\ \{B\}}{\{A\}\ x{:=}E\ ;\ C\ \{B\}}\ x' \text{ fresh}$$

This rule works in standard Hoare logic: the fact that the Floyd axiom expresses the strongest postcondition translates into its soundness and completeness. All of the rules mentioning fresh variables are obtained in this way from axioms of Separation Logic. This (standard) trick allows use of a quantifier-free language.

We will not explicitly give the semantics of commands, but assume Separation Logic's "fault-avoiding" semantics of triples (as in, e.g., [12]) in:

Theorem 1. *All of the operational rules are sound (preserving validity), and all except for* Dispose *are complete (preserving invalidity).*

To see the incompleteness of the Dispose rule consider:

$$\{x{\mapsto}[] * y{\mapsto}[]\}\ \texttt{dispose}(x)\ ;\ \texttt{empty}\ \{x{\neq}y \mid y{\mapsto}[]\}$$

This is a true triple, but if we apply the Dispose and Empty rules upwards we will be left with an entailment $y{\mapsto}[] \vdash x{\neq}y \mid y{\mapsto}[]$ that is false. The rule loses the implied information that x and y are unequal from the precondition. Although we can construct artificial examples like this that fool the rule, none of the naturally-occurring examples that we have tried in Smallfoot have suffered from it. The reason, so it seems, is that required inequalities tend to be indicated in boolean conditions in programs, in either while loops or conditionals. We have considered hack solutions to this problem but nothing elegant has arisen; so in lieu of practical problems with the incompleteness, we have opted for the simple solution presented here.

This incompleteness could be dealt with if we instead used the backwards-running weakest preconditions of Separation Logic [4]. Unfortunately, there is no existing automatic theorem prover which can deal with the form of these assertions (which use quantification and the separating implication $-\!\!*$). If there were such a prover, we would be eager consumers of it.

3.2 Rearrangement Rules

The operational rules are not sufficient on their own, because some of them expect their preconditions to be in particular forms. For instance, in

$$\{x{=}y \mid z{\mapsto}[f{:}\,w] * y{\mapsto}[f{:}\,z]\}\ x{\to}f{:=}\,y\ ;\ C\ \{\Pi' \mid \Sigma'\}$$

the Mutate rule cannot fire (be applied upwards), because the precondition has to explicitly have $x{\mapsto}[\rho]$ for some ρ.

Symbolic execution has a separate rearrangement phase, which attempts to put the precondition in the proper form for an operational rule to fire. For instance, in the example just given we can observe that the precondition $x{=}y \mid z{\mapsto}[f{:}\,w] * y{\mapsto}[f{:}\,z]$ is equivalent to $x{=}y \mid z{\mapsto}[f{:}\,w] * x{\mapsto}[f{:}\,z]$, which is in a form that allows the Mutate rule to fire.

We use notation for atomic commands that access heap cell E:

$$A(E) ::= E{\to}f{:=}\,F \mid x{:=}\,E{\to}f \mid \texttt{dispose}(E)$$

The basic rearrangement rule simply makes use of equalities to recognize that a dereferencing step is possible.

$$\textsc{Switch}(E)\quad \frac{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * F{\mapsto}[\rho]\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}\ \Pi \mid \Sigma * F{\mapsto}[\rho] \vdash E{=}F$$

For trees and list segments we have rules that expose $\mapsto$ facts by unrolling their inductive definitions, when we have enough information to conclude that the tree or the list is nonempty.[2] A nonempty tree is one that is not nil.

$$\textsc{Unroll Tree}(E)\quad \frac{\{\Pi \mid \Sigma * E{\mapsto}[l\colon x', r\colon y'] * \mathsf{tree}(x') * \mathsf{tree}(y')\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * \mathsf{tree}(F)\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}\ {}^{\dagger}$$

†when $\Pi \mid \Sigma * \mathsf{tree}(F) \vdash F{\neq}\mathsf{nil} \wedge F{=}E$ and x', y' fresh

Here, we have placed the "side condition", which is necessary for the rule to apply, below it, for space reasons. Besides unrolling the tree definition some matching is included using the equality $F{=}E$.

To unroll a list segment we need to know that the beginning and ending points are different, which implies that it is nonempty.

$$\textsc{Unroll List Segment}(E)\quad \frac{\{\Pi \mid \Sigma * E{\mapsto}[n\colon x'] * \mathsf{ls}(x', F')\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * \mathsf{ls}(F, F')\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}\ {}^{\dagger}$$

†when $\Pi \mid \Sigma * \mathsf{ls}(F, F') \vdash F{\neq}F' \wedge E{=}F$ and x' fresh

These rearrangement rules are very deterministic, and are not complete on their own. The reason is that it is possible for an assertion to imply that a cell is allocated, without knowing which $*$-conjunct it necessarily lies in. For example, the assertion $y{\neq}z \mid \mathsf{ls}(x, y) * \mathsf{ls}(x, z)$ contains a "spooky disjunction": it implies that one of the two list segments is nonempty, so that $x{\neq}y \vee x{\neq}z$, but we do not know which. To deal with this in the rearrangement phase we rely on a procedure for exorcising these spooky disjunctions. In essence, $\mathit{exor}(\Pi \mid \Sigma, E)$ is a collection of assertions obtained by doing enough case analysis (adding equalities and inequalities to Π) so that the location of E within a $*$-conjunct is determined. This makes the rearrangement rules complete.

We omit a formal definition of *exor* for space reasons. It is mentioned in the symbolic execution algorithm below, where $\mathit{exor}(g, E)$ is obtained from triple g by applying *exor* to the precondition.

3.3 Symbolic Execution Algorithm

The symbolic execution algorithm works by proof-search using the operational and rearrangement rules. Rearrangement is controlled to ensure termination.

To describe symbolic execution we presume an oracle $\mathit{oracle}(\Pi \mid \Sigma \vdash \Pi' \mid \Sigma')$ for deciding entailments. We also use that we can express consistency of a symbolic heap, and allocatedness of an expression, using entailments:

[2] This is somewhat akin to the "focus" step in shape analysis [14].

$$\mathit{incon}(\Pi \mid \Sigma) \stackrel{\text{def}}{=} \mathit{oracle}(\Pi \mid \Sigma \vdash \mathsf{nil}{\neq}\mathsf{nil} \mid \mathsf{emp})$$
$$\mathit{allocd}(\Pi \mid \Sigma, E) \stackrel{\text{def}}{=} \mathit{incon}(\Pi \mid \Sigma * E{\mapsto}[]) \text{ and } \mathit{incon}(E{=}\mathsf{nil} \wedge \Pi \mid \Sigma)$$

We also use $\mathit{pre}(g)$ to denote the precondition in a Hoare triple g. *incon* and *pre* are used to check the precondition for inconsistency in the first step of the symbolic execution algorithm and *allocd* is used in the second-last line.

Definition 2. *E is active in g if g is of the form*

$$\{\Pi \mid \Sigma\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}$$

Algorithm 3 (Symbolic Execution). *Given a triple g, determines whether or not it is provable.*

check(g) =
  if *incon*(*pre*(g)) **return** *"true"*
  if *g matches the conclusion of an operational rule*
    let *p be the premise, or p_1, p_2 the two premises* **in**
    if *rule* Empty **return** *oracle*(p)
    if *rule* Assign, Mutate, New, Dispose, *or* Lookup **return** *check*(p)
    if *rule* Conditional **return** *check*(p_1) $\wedge$ *check*(p_2)
  elseif *g begins with $A(E)$*
    if Switch(E), Unroll List Segment(E), *or* Unroll Tree(E) *applies*
      let *p be the premise* **in return** *check*(p)
    elseif *allocd*($\mathit{pre}(g), E$) **return** $\bigwedge\{\mathit{check}(g') \mid g' \in \mathit{exor}(g, E)\}$
    else return *"false"*

Theorem 4. *The Symbolic Execution algorithm terminates, and returns "true" iff there is a proof of the input judgment using the operational and rearrangement rules, where we view each use of an entailment in the symbolic execution rules as a call to the oracle.*

4 Proof Rules for Entailments

The entailment $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$ was treated as an oracle in the description of symbolic execution. We now describe a proof theory for entailment.

The rules come in two groups. The first, the normalization rules, get rid of equalities as soon as possible so that the forthcoming rules can be formulated using simple pattern matching (*i.e.*, we can use $E{\mapsto}F$ rather than $E'{\mapsto}F$ plus $E'{=}E$ derivable), make derivable inequalities explicit, perform case analysis using a form of excluded middle, and recognize inconsistency. The second group of rules, the subtraction rules, work by explicating and then removing facts from the right-hand side of an entailment, with the eventual aim of reducing to the axiom $\Pi \mid \mathsf{emp} \vdash \mathsf{true} \mid \mathsf{emp}$.

Before giving the rules, we introduce some notation. We write $\mathit{op}(E)$ as an abbreviation for $E{\mapsto}[\rho]$, $\mathsf{ls}(E, E')$, or $\mathsf{tree}(E)$. The guard $G(\mathit{op}(E))$ is defined by:

$$G(E{\mapsto}[\rho]) \stackrel{\text{def}}{=} \mathsf{true} \qquad G(\mathsf{ls}(E, E')) \stackrel{\text{def}}{=} E{\neq}E' \qquad G(\mathsf{tree}(E)) \stackrel{\text{def}}{=} E{\neq}\mathsf{nil}$$

Table 3. Proof System for Entailment

Normalization Rules:

$$\frac{}{\Pi \wedge E{\neq}E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

$$\frac{\Pi[E/x] \mid \Sigma[E/x] \vdash \Pi'[E/x] \mid \Sigma'[E/x]}{\Pi \wedge x{=}E \mid \Sigma \vdash \Pi' \mid \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge E{=}E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

$$\frac{\Pi \wedge G(op(E)) \wedge E{\neq}\mathsf{nil} \mid op(E) * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge G(op(E)) \mid op(E) * \Sigma \vdash \Pi' \mid \Sigma'}\ E{\neq}\mathsf{nil} \notin \Pi \wedge G(op(E))$$

$$\frac{\Pi \wedge E_1{\neq}E_2 \mid op_1(E_1) * op_2(E_2) * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid op_1(E_1) * op_2(E_2) * \Sigma \vdash \Pi' \mid \Sigma'}\ \begin{array}{l} G(op_1(E_1)), G(op_2(E_2)) \in \Pi \\ E_1{\neq}E_2 \notin \Pi \end{array}$$

$$\frac{\begin{array}{l}\Pi \wedge E_1{=}E_2 \mid \Sigma \vdash \Pi' \mid \Sigma' \\ \Pi \wedge E_1{\neq}E_2 \mid \Sigma \vdash \Pi' \mid \Sigma'\end{array}}{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}\ \begin{array}{l} E_1 \not\equiv E_2 \\ E_1{=}E_2, E_1{\neq}E_2 \notin \Pi \\ fv(E_1, E_2) \subseteq fv(\Pi, \Sigma, \Pi', \Sigma') \end{array}$$

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma * \mathsf{tree}(\mathsf{nil}) \vdash \Pi' \mid \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma * \mathsf{ls}(E, E) \vdash \Pi' \mid \Sigma'}$$

Subtraction Rules:

$$\frac{}{\Pi \mid \mathsf{emp} \vdash \mathsf{true} \mid \mathsf{emp}} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \wedge E{=}E \mid \Sigma'} \qquad \frac{\Pi \wedge P \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge P \mid \Sigma \vdash \Pi' \wedge P \mid \Sigma'}$$

$$\frac{S \vdash S' \qquad \Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid S * \Sigma \vdash \Pi' \mid S' * \Sigma'} \qquad \frac{}{S \vdash S} \qquad \frac{}{E{\mapsto}[\rho, \rho'] \vdash E{\mapsto}[\rho]}$$

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid \mathsf{tree}(\mathsf{nil}) * \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid \mathsf{ls}(E, E) * \Sigma'}$$

$$\frac{\Pi \mid E{\mapsto}[l{:}\, E_1, r{:}\, E_2, \rho] * \Sigma \vdash \Pi' \mid E{\mapsto}[l{:}\, E_1, r{:}\, E_2, \rho] * \mathsf{tree}(E_1) * \mathsf{tree}(E_2) * \Sigma'}{\Pi \mid E{\mapsto}[l{:}\, E_1, r{:}\, E_2, \rho] * \Sigma \vdash \Pi' \mid \mathsf{tree}(E) * \Sigma'}\ \dagger$$
$$\dagger E{\mapsto}[l{:}\, E_1, r{:}\, E_2, \rho] \notin \Sigma'$$

$$\frac{\Pi \wedge E_1{\neq}E_3 \mid E_1{\mapsto}[n{:}\, E_2, \rho] * \Sigma \vdash \Pi' \mid E_1{\mapsto}[n{:}\, E_2, \rho] * \mathsf{ls}(E_2, E_3) * \Sigma'}{\Pi \wedge E_1{\neq}E_3 \mid E_1{\mapsto}[n{:}\, E_2, \rho] * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_3) * \Sigma'}\ \dagger$$
$$\dagger E_1{\mapsto}[n{:}\, E_2, \rho] \notin \Sigma'$$

$$\frac{\Pi \mid \mathsf{ls}(E_1, E_2) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, \mathsf{nil}) * \Sigma'}{\Pi \mid \mathsf{ls}(E_1, E_2) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, \mathsf{nil}) * \Sigma'}$$

$$\frac{\Pi \wedge G(op(E_3)) \mid \mathsf{ls}(E_1, E_2) * op(E_3) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \Sigma'}{\Pi \wedge G(op(E_3)) \mid \mathsf{ls}(E_1, E_2) * op(E_3) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_3) * \Sigma'}$$

The proof rules are given in Table 3. Except for $G(op_1(E_1)), G(op_2(E_2)) \in \Pi$, the side-conditions are not needed for soundness, but ensure termination.

Theorem 5 (Soundness and Completeness). *Any provable entailment is valid, and any valid entailment is provable.*

The side-conditions are sufficient to ensure that progress is made when applying rules upwards. Decidability then follows using the naive proof procedure which tries all possibilities, backtracking when necessary.

Theorem 6 (Decidability). *Entailment is decidable.*

It is possible, however, to do much better than the naive procedure. For example, one narrowing of the search space is a phase distinction between normalization and subtraction rules: Any subtraction rule can be commuted above any normalization rule. Further commutations are possible for special classes of assertion, and these are used in Smallfoot.

This system's proof rules can be viewed as coming from certain implications, and are arranged as rules just to avoid the explicit use of the cut rule in proof search. For instance, the fourth normalization rule comes from the implications:

$$E{\mapsto}[] \to E{\neq}\mathsf{nil} \qquad\qquad E_1{\neq}E_2 \wedge \mathsf{ls}(E_1, E_2) \to E_1{\neq}\mathsf{nil}$$

the fifth from the implications:

$$E_1{\mapsto}[\rho_1] * E_2{\mapsto}[\rho_2] \to E_1{\neq}E_2 \qquad E_2{\neq}\mathsf{nil} \wedge E_1{\mapsto}[\rho] * \mathsf{tree}(E_2) \to E_1{\neq}E_2$$

$$E_2{\neq}E_3 \wedge E_1{\mapsto}[\rho] * \mathsf{ls}(E_2, E_3) \to E_1{\neq}E_2$$

$$E_1{\neq}\mathsf{nil} \wedge E_2{\neq}\mathsf{nil} \wedge \mathsf{tree}(E_1) * \mathsf{tree}(E_2) \to E_1{\neq}E_2$$

$$E_1{\neq}\mathsf{nil} \wedge E_2{\neq}E_3 \wedge \mathsf{tree}(E_1) * \mathsf{ls}(E_2, E_3) \to E_1{\neq}E_2$$

$$E_1{\neq}E_3 \wedge E_2{\neq}E_4 \wedge \mathsf{ls}(E_1, E_3) * \mathsf{ls}(E_2, E_4) \to E_1{\neq}E_2$$

and the last two from the implications:

$$\mathsf{tree}(\mathsf{nil}) \to \mathsf{emp} \qquad\qquad \mathsf{ls}(E, E) \to \mathsf{emp}$$

For the inductive predicates, these implications are consequences of unrolling the inductive definition in the metatheory. But note that we do not unroll predicates, instead case analysis via excluded middle takes one judgment to several.

Likewise, the subtraction rules for the inductive predicates are obtained from the implications:

$$\mathsf{emp} \to \mathsf{tree}(\mathsf{nil}) \qquad E{\mapsto}[l{:}\, E_1, r{:}\, E_2, \rho] * \mathsf{tree}(E_1) * \mathsf{tree}(E_2) \to \mathsf{tree}(E)$$

$$\mathsf{emp} \to \mathsf{ls}(E, E) \qquad E_1{\neq}E_3 \wedge E_1{\mapsto}[n{:}\, E_2, \rho] * \mathsf{ls}(E_2, E_3) \to \mathsf{ls}(E_1, E_3)$$

$$\mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, \mathsf{nil}) \to \mathsf{ls}(E_1, \mathsf{nil})$$

$$\mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * E_3{\mapsto}[\rho] \to \mathsf{ls}(E_1, E_3) * E_3{\mapsto}[\rho]$$

$$E_3{\neq}\mathsf{nil} \wedge \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \mathsf{tree}(E_3) \rightarrow \mathsf{ls}(E_1, E_3) * \mathsf{tree}(E_3)$$

$$E_3{\neq}E_4 \wedge \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \mathsf{ls}(E_3, E_4) \rightarrow \mathsf{ls}(E_1, E_3) * \mathsf{ls}(E_3, E_4)$$

The first four are straightforward, while the last four express properties whose verification of soundness would use inductive proofs in the metatheory. The resulting rules do not, however, require a search for inductive premises. In essence, what we generally do is, for each considered inductive predicate, add a collection of rules that are consequences of induction, but that can be formulated in a way that preserves the proof theory's terminating nature.

In the last subtraction rule, the $G(op(E_3)) \wedge op(E_3)$ part of the left-hand side ensures that E_3 does not occur within the segments from E_1 to E_2 or from E_2 to E_3. This is necessary for appending list segments, since they are required to be acyclic.

Here is an example proof, of the entailment mentioned in the Introduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{t{\neq}\mathsf{nil} \mid \mathsf{emp} \vdash \mathsf{emp}}}{t{\neq}\mathsf{nil} \mid \mathsf{ls}(y, \mathsf{nil}) \vdash \mathsf{ls}(y, \mathsf{nil})}}{t{\neq}\mathsf{nil} \mid t{\mapsto}[n\colon y] * \mathsf{ls}(y, \mathsf{nil}) \vdash t{\mapsto}[n\colon y] * \mathsf{ls}(y, \mathsf{nil})}}{t{\neq}\mathsf{nil} \mid t{\mapsto}[n\colon y] * \mathsf{ls}(y, \mathsf{nil}) \vdash \mathsf{ls}(t, \mathsf{nil})}}{t{\neq}\mathsf{nil} \mid \mathsf{ls}(x, t) * t{\mapsto}[n\colon y] * \mathsf{ls}(y, \mathsf{nil}) \vdash \mathsf{ls}(x, \mathsf{nil})}}{\mathsf{ls}(x, t) * t{\mapsto}[n\colon y] * \mathsf{ls}(y, \mathsf{nil}) \vdash \mathsf{ls}(x, \mathsf{nil})}$$

Going upwards, this applies the normalization rule which introduces $t{\neq}\mathsf{nil}$, then the subtraction rule for nil-terminated list segments, the subtraction rule for nonempty list segments, and finally $*$-Introduction (the basic subtraction rule for $*$, which appears fourth) twice.

5 Incomplete Proofs and Frame Axioms

Typically, at a call site to a procedure the symbolic heap will be larger than that required by the procedure's precondition. This is the case in the disp_tree example where, for example, the symbolic heap at one of the recursive call sites is $p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$, where that expected by the procedure specification of disp_tree(i) is just $\mathsf{tree}(i)$. We show how to use the proof theory from the previous section to infer frame axioms.

In more detail the (spatial) part of the problem is,

> *Given*: two symbolic heaps, $\Pi \mid \Sigma$ (the heap at the call site), and $\Pi_1 \mid \Sigma_1$ (the procedure precondition)
> *To Find*: a spatial predicate Σ_F, the "spatial frame axiom", satisfying the entailment $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F$.

Our strategy is to search for a proof of the judgment $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1$, and if this search, going upwards, halts at $\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}$ then Σ_F is a sound choice as a frame axiom. We give a few examples to show how this mechanism works.

First, and most trivially, let us consider the disp_tree example:

Assertion at Call Site : $p{\mapsto}[l{:}\,i, r{:}\,j] * \mathsf{tree}(i) * \mathsf{tree}(j)$

Procedure Precondition : $\mathsf{tree}(i)$

Then an instance of $*$-Introduction

$$\frac{p{\mapsto}[l{:}\,i, r{:}\,j] * \mathsf{tree}(j) \vdash \mathsf{emp}}{p{\mapsto}[l{:}\,i, r{:}\,j] * \mathsf{tree}(i) * \mathsf{tree}(j) \vdash \mathsf{tree}(i)}$$

immediately furnishes the correct frame axiom: $p{\mapsto}[l{:}\,i, r{:}\,j] * \mathsf{tree}(j)$.

For an example that requires a little bit more logic, consider:

Assertion at Call Site : $x{\mapsto}[] * y{\mapsto}[]$

Procedure Precondition : $x{\neq}y \mid x{\mapsto}[]$

$$\frac{\dfrac{\dfrac{x{\neq}y \mid y{\mapsto}[] \vdash \mathsf{emp}}{x{\neq}y \mid y{\mapsto}[] \vdash x{\neq}y \mid \mathsf{emp}}}{x{\neq}y \mid x{\mapsto}[] * y{\mapsto}[] \vdash x{\neq}y \mid x{\mapsto}[]}}{x{\mapsto}[] * y{\mapsto}[] \vdash x{\neq}y \mid x{\mapsto}[]}$$

Here, the inequality $x{\neq}y$ is added to the left-hand side in the normalization phase, and then it is removed from the right-hand side in the subtraction phase.

On the other hand, consider what happens in a wrong example:

Assertion at Call Site : $x{\mapsto}[] * y{\mapsto}[]$

Procedure Precondition : $x{=}y \mid x{\mapsto}[]$

$$\frac{\dfrac{\dfrac{??}{x{\neq}y \mid y{\mapsto}[] \vdash x{=}y \mid \mathsf{emp}}}{x{\neq}y \mid x{\mapsto}[] * y{\mapsto}[] \vdash x{=}y \mid x{\mapsto}[]}}{x{\mapsto}[] * y{\mapsto}[] \vdash x{=}y \mid x{\mapsto}[]}$$

In this case we get stuck at an earlier point because we cannot remove the equality $x{=}y$ from the right-hand side in the subtraction phase. To correctly get a frame axiom we have to obtain true in the pure part of the right-hand side; we do not do so in this case, and we rightly do not find a frame axiom.

The proof-theoretic justification for this method is the following.

Theorem 7. *Suppose that we have an incomplete proof (a proof that doesn't use axioms):*

$$\begin{array}{c}[\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}] \\ \vdots \\ \Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1\end{array}$$

Then there is a complete proof (without premises, using an axiomatic rule at the top) of:

$$\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F.$$

This justifies an extension to the symbolic execution algorithm. In brief, we extend the syntax of loop-free triples with a **jsr** instruction

$$C ::= \cdots \mid [\Pi \mid \Sigma]\ \mathbf{jsr}\ [\Pi' \mid \Sigma']\,; C \quad \text{jump to subroutine}$$

annotated with a precondition and a postcondition. In Smallfoot this is generated when an annotated program is chopped into straightline Hoare triples. The appropriate operational rule is:

$$\frac{\Pi \mid \Sigma \vdash \Pi_1 \wedge \Pi \mid \Sigma_1 * \Sigma_F \qquad \{\Pi_2 \wedge \Pi \mid \Sigma_2 * \Sigma_F\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ [\Pi_1 \mid \Sigma_1]\ \mathbf{jsr}\ [\Pi_2 \mid \Sigma_2]\,;C\ \{\Pi' \mid \Sigma'\}}$$

When we encounter a **jsr** command during symbolic execution we run the proof theory from the previous section upwards with goal $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1$. If it terminates with $\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}$ then we tack Σ_F onto the postcondition Σ_2, and we continue execution with C. Else we report an error.

The description here is simplified. Theorem 7 only considers incomplete proofs with single assumptions, but it is possible to generalize the treatment of frame inference to proofs with multiple assumptions (which leads to several frames being checked in symbolic execution). Also, we have only discussed the spatial part of the frame, neglecting modifies clauses for stack variables. A pure frame must also be discovered, but that is comparatively easy.

Finally, this way of inferring frame axioms works, but is incomplete. To see why, for $[x{\mapsto}-]$ **jsr** [emp] we run into a variant of the same problem discussed before in incompleteness of the `dispose` instruction: it we added a frame $y{\mapsto}-$ then the postcondition would lose the information that $x{\neq}y$. Similar incompleteness arises for larger-scale operations as well, such as disp_tree. Now, the incompleteness is not completely disastrous. When reasoning about recursive calls to disp_tree, never do we need to conclude an inequality between, say, a just-disposed cell in the left subtree and a cell in the right; $*$ gives us the information we need, at the right time, for the proof to go through.

It is an incompleteness, still.

6 Conclusion

The heap poses great problems for modular verification and analysis. For example, PALE is (purposely) unsound in its treatment of frame axioms for procedures [7], the "modular soundness" of ESC is subtle but probably not definitive [6], interprocedural Shape Analysis is just beginning to become modular [13].

We believe that symbolic execution with Separation Logic has some promise in this area. An initial indication is the local way that heap update is treated in symbolic execution: there is no need to traverse an entire heap structure when an update to a single cell is done, as is the case with Shape Analysis [14]. Going beyond this initial point, it will be essential to have a good way of inferring frame axioms. We have sketched one method here, but there are likely to be others, and what we have done is only a start.

There are similarities between this work and the line of work started by Alias Types [16], however there are crucial differences. One of the most significant points is that here we (completely) axiomatize the consequences of induction for our inductive predicates, while the 'coercions' of [16] include only rolling and unrolling of inductive definitions. Relatedly, here we capture semantic entailment between formulæ exactly, as opposed to providing a coarse approximation. Additionally, this enables commands to branch on possibly inductive consequences of heap shape. Another crucial difference is that here we rely on Separation Logic's

Frame Rule for a very strong form of modularity, and infer frame axioms using incomplete proofs, while Alias Types uses second-order quantification (store polymorphism) with manual instantiation. These differences aside, one wonders whether the lines of work stemming from Alias Types and Separation Logic will someday merge; an interesting step along these lines is in [8], and we are investigating uses of bunched typing [11, 9] for similar purposes.

A different way of automating Separation Logic has recently been put forward by Jia and Walker [5]. An interesting part of their system is how classical arithmetic and substructural logic work together. They also provide a decidable fragment based on Linear Logic. There is a gap between the entailment of their proof theory and that of the heap model, because Linear Logic's proof theory is purposely incomplete for the standard additives supported by the model.

To build on this paper's formulation of symbolic heaps, we would particularly like to have a general scheme of inductive definitions rather than using hardwired predicates. (We are not just asking for semantically well-defined recursive predicates, *e.g.*, as developed in [15], but would want a, hopefully terminating, proof theory.) Soundly extending the techniques here to a class of inductive predicates which generalizes those presented is largely straightforward, since the operational symbolic execution rules would be unaffected, the necessary rearrangement rule for unrolling a more general inductive predicate depends on a certain shape of the inductive definitions where unrolling is triggered by inequalities, and the proof system for entailment would remain sound. Maintaining the present degree of completeness, on the other hand, is nontrivial, since the proof system for entailment becomes incomplete, and exorcising spooky disjunctions may become incomplete (that is, modifying *exor* such that the 'if' direction of Theorem 4 holds is (very) hard).

We would like to relax the restriction to quantifier-free assertions. For example, with $\exists y.\, x{\mapsto}[n\colon y] * \mathsf{ls}(y, x)$ we can describe a circular linked list that has at least one element. It may be that a restricted amount of existential quantification is compatible with having a complete and terminating proof theory.

We should admit that consideration of completeness has greatly slowed us down in this work. The main ideas in this paper were present, and implemented in an early version of Smallfoot, over two years ago. But, the third author (perhaps foolishly) then asked the first two: Is your proof theory complete? And if not, please give an undecidability result, thus rendering completeness impossible. Now, we know that completeness is an ideal that will not always be possible to achieve, but the first two authors were eventually able to answer in the affirmative. Although an ideal, we stress that we would not be satisfied with a sound proof theory based (just) on rolling and unrolling definitions. Having a mechanism to provide axioms for consequences of induction when defining inductive predicates is essential. Without such axioms, it is not possible to verify programs that work at the end of a singly-linked list, or at both ends of a doubly-linked list (we have given similar proof rules for both conventional doubly-linked and xor-linked lists).

That being said, our symbolic execution mechanism is incomplete in two other areas, the treatment of disposal and inference of frame axioms. The former is perhaps reparable by hacks, but the latter is more fundamental.

The ideas in this paper would seem to provide a basis for investigating program analysis. The first crucial question is the exact nature of the abstract domain of formulae, which would enable the calculation of invariants by fixed-point approximation. After that, we would like to attack the problem of modular, interprocedural heap analysis, leveraging the strong modularity properties of Separation Logic.

Acknowledgements. We are grateful to the anonymous referees for helpful comments. During Berdine's stay at Carnegie Mellon University during 2003, this research was sponsored in part by National Science Foundation Grant CCR-0204242. All three authors were supported by the EPSRC.

References

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS 2004*, volume 3328 of *LNCS*, pages 97–109. Springer, Dec. 2004.
[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: A tool for checking Separation Logic footprint specifications. In preparation, 2005.
[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, 1977.
[4] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL '01*, pages 14–26, 2001.
[5] L. Jia and D. Walker. ILC: A foundation for automated reasoning about pointer programs. Draft., Apr. 2005.
[6] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.
[7] A. Möller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01*, pages 221–231, 2001.
[8] G. Morrisett, A. J. Ahmed, and M. Fluet. L^3: A linear language with locations. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *LNCS*, pages 293–307, 2005.
[9] P. W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.
[10] 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.
[11] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.
[12] J. C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.
[13] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL '05*, pages 296–309, 2005.
[14] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
[15] É.-J. Sims. Extending separation logic with fixpoints and postponed substitution. In *AMAST*, volume 3116 of *LNCS*, pages 475–490. Springer, 2004.
[16] D. Walker and J. G. Morrisett. Alias types for recursive data structures. In *Types in Compilation*, volume 2071 of *LNCS*, pages 177–206, 2001.