

Smallfoot: Modular Automatic Assertion Checking with Separation Logic

Josh Berdine¹, Cristiano Calcagno², and Peter W. O’Hearn³

¹ Microsoft Research

² Imperial College, London

³ Queen Mary, University of London

Abstract. Separation logic is a program logic for reasoning about programs that manipulate pointer data structures. We describe Smallfoot, a tool for checking certain lightweight separation logic specifications. The assertions describe the shapes of data structures rather than their detailed contents, and this allows reasoning to be fully automatic. The presentation in the paper is tutorial in style. We illustrate what the tool can do via examples which are oriented toward novel aspects of separation logic, namely: avoidance of frame axioms (which say what a procedure does not change); embracement of “dirty” features such as memory disposal and address arithmetic; information hiding in the presence of pointers; and modular reasoning about concurrent programs.

1 Introduction

Separation logic is a program logic geared toward specifying and verifying properties of dynamically-allocated linked data structures [35], which has led to much simpler by-hand specifications and program proofs than previous formalisms. Specifications in separation logic are “small”, in that a specification of a program component concentrates on the resources relevant to its correct operation (its “footprint”), not mentioning the resources of other components at all [32]. In this paper we describe Smallfoot, an experimental tool for checking certain separation logic specifications.

The aim of the tool was simple: we wanted to see whether the simplicity of the by-hand proofs in separation logic could be transferred to an automatic setting. Smallfoot uses lightweight assertions that describe the shapes of data structures rather than their detailed contents; this restriction allows the reasoning to be fully automatic. The input language allows first-order procedures with reference and value parameters, essentially as in [17], together with operations for allocating, deallocating, mutating and reading heap cells. Smallfoot requires pre- and post-conditions for the procedures, and loop invariants. It also supports annotations for concurrency, following a concurrent extension of separation logic [31, 11].

In [5] we defined the symbolic execution mechanism and proof procedure that lie at the heart of Smallfoot, but we did not there show how they could be used to prove programs. The purpose of this paper is the opposite: to show what the

tool can do, without exposing its innards. We proceed in a tutorial style. We describe in an informal way how the proof rules of [32, 35, 31] are used in the tool, in conjunction with the execution mechanism, but we do not give a fully formal description or a repeat of the techniques of [5]. For a full understanding of exactly how Smallfoot works familiarity with [32, 35, 31, 5] is essential. But we have tried to make the presentation relatively self-contained, and we hope that many of the main points can be gleaned from our examples and the discussion surrounding them.

We begin in the next section by introducing Smallfoot with three examples. The purpose of this work is to explore separation logic’s modularity in an automatic setting, and that is the subject of all three examples. We will discuss some of the features of Smallfoot as we go through the examples, and highlight some of the issues for automation that guided its design. A description of the input language and some central points in the verification condition mechanism is then given in Sections 3 and 4. Several further examples are given in Section 5, and we conclude with a discussion of related and future work.

We stress that Smallfoot is limited in various ways. Its input language has been designed to match the theoretical work on separation logic, rather than an existing widely-used language; our purpose was to experiment with the logic, rather than to produce a mature end-user tool. Beyond the basic primitives of separation logic, Smallfoot at this point includes several hardwired predicates for singly-, doubly-, and xor-linked lists, and for trees, but not (yet) a mechanism for arbitrary inductive definitions of data structures. We included xor lists just to illustrate how reachability does not feature in separation logic; we have not incorporated more general address arithmetic. Smallfoot cannot handle all of the more advanced algorithms that have been the subject of by-hand proofs in separation logic, particularly graph algorithms [40, 6, 7]. Further, it does not have specifications for full functional correctness. Extensions in some of these directions would necessitate abandoning the automatic aspect, relying on interactive proof. Those are areas for further work.

2 Smallfoot in Three Nutshells

We begin with a warning: you should suspend thinking about the global heap when reading separation logic specifications, otherwise the logic can seem counterintuitive. Rather than global heaps you can think of heaplets, portions of heap. An assertion talks about a heaplet rather than the global heap, and a spec $[P]C[Q]$ says that if C is given a heaplet satisfying P then it will never try to access heap outside of P (other than cells allocated during execution) and it will deliver a heaplet satisfying Q if it terminates. (Of course, this has implications for how C acts on the global heap.) This heaplet reading may seem a simple point, but we have found that separation logic’s “local way of thinking” can lead to confusions, which arise from reverting to thinking in terms of the global heap. So we will return to this point several times below.

2.1 Local Specifications and Framing

Consider a procedure for disposing a tree:

```
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]
```

This is the expected procedure that walks a tree, recursively disposing left and right subtrees and then the root pointer. It uses a representation of tree nodes with left and right fields, and the empty tree is represented by nil.

This Smallfoot program includes a precondition and postcondition, corresponding to a partial correctness specification:

$$[\text{tree}(p)] \text{ disp_tree}(p) [\text{emp}]$$

(We use square instead of curly brackets, despite treating partial correctness, to maintain consistency with Smallfoot’s concrete syntax.) This is an example of the small specifications supported by separation logic: it talks only about the portion of heap relevant to the correct operation of the procedure. In particular, $\text{tree}(p)$ describes a heaplet where p points to a tree, *and where there are no junk cells, cells not in the tree*. This “no junk cells” part is necessary to be able to conclude emp , that the heaplet on termination is empty.

Smallfoot discovers a proof of this program by symbolic execution. The proof in the else branch corresponds to the proof steps:

$$\begin{aligned} & [(p \rightarrow l: x, r: y) * \text{tree}(x) * \text{tree}(y)] \\ & \quad i := p \rightarrow l; j := p \rightarrow r; \\ & [(p \rightarrow l: i, r: j) * \text{tree}(i) * \text{tree}(j)] \\ & \quad \text{disp_tree}(i); \\ & [(p \rightarrow l: i, r: j) * \text{tree}(j)] \\ & \quad \text{disp_tree}(j); \\ & [p \rightarrow l: i, r: j] \\ & \quad \text{dispose}(p); \\ & [\text{emp}] \end{aligned}$$

After we enter the else branch we know that $p \neq \text{nil}$ so that, by unrolling, p is an allocated node that points to left and right subtrees occupying separate storage. Then the roots of the two subtrees are loaded into i and j . Notice how the next proof steps follow operational intuition. The first recursive call removes the left subtree, the second call removes the right subtree, and the final instruction removes the root pointer p . The occurrences of the separating conjunction $*$ in these assertions ensure that the structures described, the two subtrees and root pointer, occupy separate memory, as is necessary if an operation that removes one of them is not to affect one of the others. This verification is carried out using the specification of disp_tree as an assumption, as in the usual treatment of recursive procedures in Hoare logic [17].

In the if branch we use an implication $\text{tree}(p) \wedge p = \text{nil} \Rightarrow \text{emp}$, which relies on the “no junk” character of the `tree` predicate.

The assertions in this proof use very little of separation logic; they are all of the form $\Pi \wedge \Sigma$ where Π is a pure boolean condition and Σ is a $*$ -combination of heap predicates. All of the assertions in `Smallfoot` are of this special form (together with conditionals over them), and this enables a symbolic execution mechanism where $*$ -conjuncts are updated in-place.

There is a hidden part in the proof outline just given: in the two procedure calls the preconditions at the call sites do not match the preconditions for the overall specification of `disp_tree`. For example, for the second call the assertion at the call site is $(p \mapsto l: i, r: j) * \text{tree}(j)$ while the procedure spec would suggest that the precondition should just be $\text{tree}(j)$ (after renaming of the parameter). This is where the local way of thinking comes in. The specification of `disp_tree` says that a heaplet satisfying $\text{tree}(j)$ is transformed into one satisfying emp . The input heaplet need not be the whole heap, we can effect this transformation on a heaplet that lives inside a larger heap, and then slot the result into that larger heap.

In separation logic, this pulling out and slotting in is described using the $*$ connective. Generally, a heaplet h satisfies $P * Q$ if it can be split into two disjoint heaplets h_P and h_Q that satisfy P and Q . The above narrative for the call `disp_tree(j)` tells us to take $(p \mapsto l: i, r: j) * \text{tree}(j)$, pull out the heaplet description $\text{tree}(j)$, transform it to emp , and slot that back in, obtaining $(p \mapsto l: i, r: j) * \text{emp}$. Then, we can use an identity $P * \text{emp} \Leftrightarrow P$.

Separation logic has an inference rule (the frame rule)

$$\frac{[P] C [Q]}{[R * P] C [R * Q]}$$

(where C doesn’t assign to R ’s free variables) which lets us do “pull out, perform local surgery, slot in” in a proof. To automatically generate proofs using this rule, which was implicitly applied in the steps in the proof for the else branch above, we need a way to infer frame axioms. If we are given an assertion at a call site and a procedure precondition, we must find the leftover part (which lets us do the “pull out” step). Often, this leftover part can be found by simple pattern matching, as is the case in the `disp_tree` example, but there are other cases where pattern matching will not do. Technically, `Smallfoot` uses a method of extracting frame axioms from incomplete proofs in a proof theory for entailments [5].

2.2 Processes that Mind Their Own Business

Concurrent separation logic [31] has the following rule for parallel composition:

$$\frac{[P] C [Q] \quad [P'] C' [Q']}{[P * P'] C \parallel C' [Q * Q']}$$

where C does not change variables free in P', C', Q' , and vice versa. The idea of this rule is that the specifications $[P] C [Q]$ and $[P'] C' [Q']$ describe all the resources that C and C' might access, that they mind their own business; so,

if we know that the resources are separate in the precondition, then we can reason about the concurrent processes independently. A simple example of this is a parallel composition of two heap alterations on different cells, where the $*$ in the precondition guarantees that x and y are not aliases:

$$\begin{array}{c}
 [x \mapsto c: 3 * y \mapsto c: 3] \\
 [x \mapsto c: 3] \quad \parallel \quad [y \mapsto c: 3] \\
 x \rightarrow c := 4 \quad \parallel \quad y \rightarrow c := 5 \\
 [x \mapsto c: 4] \quad \parallel \quad [y \mapsto c: 5] \\
 [x \mapsto c: 4 * y \mapsto c: 5]
 \end{array}$$

The local thinking is exercised more strongly in concurrent than in sequential separation logic. A points-to fact $x \mapsto c: 3$ describes a heaplet with a single cell x that is a record with a c field whose value is 3. As far as the left process is concerned, reasoning is carried out for a heaplet with a single cell, its heaplet, and similarly for the right. In the global heap, though, it is not the case that there is only one cell; there are at least two! The two views, local and more global, are reconciled by the form of the concurrency rule.

To apply the concurrency rule automatically we need a way to get our hands on the preconditions of the constituent processes. We could do this in several ways, such as by requiring an annotation with each \parallel , or by introducing a “named process” concept which requires a precondition but no postcondition. We settled on requiring the constituents of a \parallel to be procedure calls; because procedures come with pre/post specs we can use their preconditions when applying the concurrency rule. The postconditions are not strictly necessary for automating the concurrency rule. We made this choice just to avoid multiplying annotation forms. A Smallfoot program corresponding to the above example, but where we create the two separate cells, is:

```

upd(x, y) [x ↦] {x → c := y;} [x ↦ c: y]

main() {
  x := new(); y := new(); x → c := 3; y → c := 3;
  upd(x, 4) || upd(y, 5);
} [x ↦ c: 4 * y ↦ c: 5]

```

In the precondition of `upd` the assertion $x \mapsto$ indicates that x points to something. It denotes a singleton heaplet in which x is the only allocated or defined cell. The postcondition describes a singleton heaplet where the c field of location x has y as its contents.

When a pre- or post-condition is left out, as the pre for `main` is in this program, it defaults to `emp`. Also, Smallfoot accepts a collection of procedures as input, one optionally “main”.

In contrast, when we change the main program to

```

main() {
  x := new(); x → c := 3; y := x;
  upd(x, 4) || upd(y, 4);
} [y = x ∧ x ↦ c: 4]

```

then Smallfoot flags an error; since x and y are aliases, there is no way to split the heap into two parts, giving one symbolic cell to each of the constituent processes. In general, if a Smallfoot program has a race — where two processes may attempt to access the same cell at the same time — then an error is reported. (More precisely, any such attempted parallel accesses must be wrapped in critical sections which specify atomicity assumptions for accesses.)

Our description of how the proof is found for sequential `disp_tree` is almost the same for a parallel variant, which Smallfoot proves using the concurrency rule:

```

par_disp_tree(p) [tree(p)] {
  local i,j;
  if (p = nil) {} else {
    i := p→l; j := p→r;
    par_disp_tree(i) || par_disp_tree(j);
    dispose(p); }
} [emp]

```

The reader’s reaction to `disp_tree` and `par_disp_tree` might be: aren’t they rather trivial? Well, yes, and that is part of the point. For contrast, consider `par_disp_tree` in the rely/guarantee formalism [21, 27], which is rightly celebrated for providing compositional reasoning about concurrency. In addition to a precondition and a postcondition saying that the nodes in the tree are deallocated, we would have to formalize two additional assertions:

Rely No other process touches my tree `tree(p)`; and
Guarantee I do not touch any storage outside my tree.

Although compositional, as this example demonstrates the relies and guarantees can be rather global, and can complicate specifications even in simple examples when no interference is present. The Smallfoot specification for this procedure is certainly simpler.

2.3 Process Interaction and Heaplet Transfer

Process interaction in Smallfoot is done with conditional critical regions (CCRs) [18]. The programming model is based on “resources” r and CCR statements `with r when(B) {C}`. CCRs for common resource r must be executed with mutual exclusion, and each has a guard which must hold before execution.

Data abstractions can be protected with CCRs by wrapping critical regions around code that accesses a data structure. A more daring form of concurrency is when several processes access the same piece of state outside of critical sections [31]. In separation logic it is possible to show that daring programming idioms are used consistently. An example is a pointer-transferring buffer: instead of copying a (perhaps large) portion of data from one process to another, a pointer to the data is sent. Typically, the sending and receiving processes access the pointer without synchronization.

A toy version of this scenario is the following code snippet using buffer operations `put` and `get`:

```

x := new();   ||   get(y);
put(x);       ||   dispose(y);

```

This creates a new pointer in the left process and then places it in the buffer. The right process then reads out the pointer and disposes it. We would typically want to fill the pointer contents in the left process before sending it, and to do something with those contents in the right. The point is that to reason about the `dispose` in the right process we must know that `y` is not dangling after we do the `get` operation. It is useful to use the intuition of “permission to access” to describe this [9, 8]: the permission to access the pointer moves from the first to the second process along with the pointer value. Further, when permission transfers it must disappear from the left process or else we could mistakenly justify a further `dispose(x)` in the left process, after the `put`. In conjunction with the `dispose(y)` in the right process that would disastrously lead to a double-disposal that we must rule out.

This is where the local way of thinking helps. An assertion at a program point describes a heaplet, which represents a local permission to access, instead of a global heap. `put(x)` will have precondition $x \mapsto$ and postcondition `emp`, the idea being that the heaplet for `x` flows out of the left process and into the buffer. The `emp` postcondition ensures that, even if the value of `x` remains unchanged, the local knowledge that `x` is not dangling (the permission) is given up, thus preventing further disposal. At this point the global heap is not empty, but the heaplet/permission for the left process is. `get(y;)` will have precondition `emp` and postcondition $y \mapsto$, connoting that the heaplet (the permission) materializes in the second process.

A Smallfoot program encoding this scenario is:

```

resource buf (c) [if c=nil then emp else c↦]
init() { c := nil; }
put(x) [x↦] { with buf when(c=nil) { c := x; } } [emp]
get(y;) [emp] { with buf when(c≠nil) { y := c; c := nil; } } [y↦]
putter() { x := new(); put(x); putter(); }
getter() { get(y;); /* use y */ dispose(y); getter(); }
main() { putter() || getter(); }

```

In the CCR model resource names are used to determine units of mutual exclusion. Different CCRs `with r when(B) {C}` for the same resource name `r` cannot overlap in their executions. A CCR can proceed with its body `C` only when its boolean condition `B` holds. A resource declaration indicates some private variables associated with the resource (in this case `c`) and an invariant that describes its internal state.

When we have resource declarations as here an `init` procedure is needed for initialization; when we do not have a resource declaration, the initialization can be omitted. The `init` procedure is run before `main`; it's job is to set up the state that is protected by the named resource, by establishing the resource invariant.

In this code the omitted preconditions and postconditions are all (by default) `emp`, except the post of `init` which is (by default) the resource invariant (the assertion in the resource declaration). The `put` and `get` operations are encoded using little critical regions. The resource `buf` has an invariant which describes its heaplet: it says that if `c=nil` then the buffer has no permission, else it holds permission to access `c`. The `put` operation can fire only when `c=nil`, and so because of the invariant we will know at that point that `buf`'s heaplet is `emp`. The assignment `c:=x` changes the `buf` state so that the only way for the invariant to be true is if `c→`; the permission to access the pointer (at this point denoted by both `c` and `x`) flows into the buffer. Furthermore, the `put` operation cannot have `x→` as its postcondition because separation is maintained between the resource invariant and the heaplet assertions for the two processes. A similar narrative can be given about how `get` effects a transfer from the buffer to the `getter` process.

In fact, the annotations in this code are more than is strictly needed. If we were to inline `put` and `get`, then Smallfoot would verify the resulting code. We separated out these operations only to display what their specifications are.

What makes this all work is an inference rule

$$\frac{[(P * R_r) \wedge B] C [Q * R_r]}{[P] \text{with } r \text{ when}(B) \{C\} [Q]}$$

where R_r is an invariant formula associated with resource r . This rule is used to verify the `put` and `get` procedures, and the concurrency rule is then used for the composition of `putter` and `getter`. Even though the program loops, the fact that it gets past Smallfoot ensures that no pointer is ever disposed twice (without an intervening allocation), that there is no race condition, and that the resource invariant is true when not in the middle of a critical section.

Besides the separation between resource and process enforced using $*$, this rule (which stems originally from [18]) is wonderfully modular: the precondition and postcondition P and Q of a CCR do not mention the invariant R_r at all. This allows reasoning about processes in isolation, even in the presence of interaction.

3 The Input Language

3.1 Annotated Programs

A Smallfoot program consists of sets of resource declarations

$$\text{resource } r(\vec{x}_r)R_r$$

where \vec{x}_r and R_r are resource r 's protected variables and invariant; and procedure declarations

$$f(\vec{p}; \vec{v})[P_f] C_f [Q_f]$$

where procedure f 's parameters \vec{p} are passed by reference and \vec{v} by value, and assertions P_f and Q_f are f 's pre- and post-conditions. In this formal description the preconditions and postconditions have to be included, but we repeat that in the tool if a pre or post is left out then it is **emp** by default. Assertions are described later; commands are generated by:

$$\begin{aligned}
E &::= x \mid \text{nil} \mid c \mid E \text{ xor } E \\
B &::= E=E \mid E \neq E \\
S &::= x:=E \mid x:=E \rightarrow t \mid E \rightarrow t:=E \mid x:=\text{new}() \mid \text{dispose}(E) \\
C &::= S \mid C;C \mid \text{if}(B) \{C\} \text{ else } \{C\} \mid \text{while}(B) [I] \{C\} \\
&\quad \mid f(\vec{x}; \vec{E}) \mid f(\vec{x}; \vec{E}) \parallel f(\vec{x}; \vec{E}) \mid \text{with } r \text{ when}(B) \{C\}
\end{aligned}$$

There is the additional evident constraint on a program that in any procedure call $f(\vec{y}; \vec{E})$ or region **with** r **when**(B) $\{C\}$ the variable f/r must be defined in a procedure/resource declaration.

Smallfoot programs are subject to certain variable restrictions, which are needed for the soundness of Hoare logic rules; for example, that variable aliasing and concurrent races for variables (not heap cells) are ruled out. These conditions are, in general, complex and unmemorable; they may be found in [4].

3.2 Assertions and Specifications

The assertions are $*$ -combinations of heap predicates and \wedge -combinations of pure boolean facts, together with conditionals over these. Conditionals are used rather than disjunctions because they preserve the “preciseness” property that is needed for soundness of concurrent separation logic [11]. The heap predicates include the points-to relation, the tree predicate, a predicate for singly-linked list segments and one for xor-linked lists. (We also have conventional doubly-linked lists in Smallfoot, but do not include any examples for them in this paper.)

$$\begin{aligned}
P, Q, R, I &::= \Pi \wedge \Sigma \mid \text{if } B \text{ then } P \text{ else } P & H &::= E \mapsto \rho \mid \text{tree}(E) \mid \text{ls}(E, E) \\
\Pi &::= B_1 \wedge \dots \wedge B_n \mid \text{true} \mid \text{false} & & \mid \text{xlseg}(E, E, E, E) \\
\Sigma &::= H_1 * \dots * H_n \mid \text{emp} & \rho &::= t_1:E_1, \dots, t_n:E_n
\end{aligned}$$

The model assumes a finite collection **Fields** (from which the t_i are drawn), and disjoint sets **Loc** of locations and **Values** of non-addressable values, with $\text{nil} \in \text{Values}$. We then set:

$$\begin{aligned}
\text{Heaps} &\stackrel{\text{def}}{=} \text{Loc} \xrightarrow{\text{fin}} (\text{Fields} \rightarrow \text{Values} \cup \text{Loc}) \\
\text{Stacks} &\stackrel{\text{def}}{=} \text{Variables} \rightarrow \text{Values} \cup \text{Loc}
\end{aligned}$$

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.

For $s \in \text{Stacks}$, $h \in \text{Heaps}$, the key clauses in the satisfaction relation for assertions are as follows:

$$\begin{array}{ll}
s \models E=F & \stackrel{\text{def}}{\text{iff}} \llbracket E \rrbracket s = \llbracket F \rrbracket s \\
s \models E \neq F & \stackrel{\text{def}}{\text{iff}} \llbracket E \rrbracket s \neq \llbracket F \rrbracket s \\
s \models II_0 \wedge II_1 & \stackrel{\text{def}}{\text{iff}} s \models II_0 \text{ and } s \models II_1 \\
s, h \models E_0 \mapsto t_1: E_1, \dots, t_k: E_k & \stackrel{\text{def}}{\text{iff}} h = \llbracket [E_0] s \rightarrow r \rrbracket \text{ where } r(t_i) = \llbracket E_i \rrbracket s \text{ for } i \in 1..k \\
s, h \models \text{emp} & \stackrel{\text{def}}{\text{iff}} h = \emptyset \\
s, h \models \Sigma_0 * \Sigma_1 & \stackrel{\text{def}}{\text{iff}} \exists h_0, h_1. h = h_0 * h_1 \text{ and } s, h_0 \models \Sigma_0 \text{ and } s, h_1 \models \Sigma_1 \\
s, h \models II \wedge \Sigma & \stackrel{\text{def}}{\text{iff}} s \models II \text{ and } s, h \models \Sigma
\end{array}$$

For pure assertions II we do not need the heap component in the satisfaction relation. $h = h_0 * h_1$ indicates that the domains of h_0 and h_1 are disjoint, and that h is their graph union. The semantics $\llbracket E \rrbracket s \in \text{Values}$ of expressions is as expected. We will not provide semantic definitions of the predicates for trees and lists now, but give inductive characterizations of them later.

Each command C determines a relation:

$$\llbracket C \rrbracket: (\text{Stacks} \times \text{Heaps}) \longleftrightarrow (\text{Stacks} \times \text{Heaps}) \cup \{\text{fault}\}$$

The fault output occurs when a command attempts to dereference a dangling pointer. For example, $x \rightarrow tl := y$ produces a fault when applied to state s, h , where $s(x)$ is not a location in the domain of h . We will not give a formal definition of $\llbracket C \rrbracket$; when considering concurrency it is especially intricate [11]. The interpretation of Hoare triples is:

$[P] C [Q]$ holds if, whenever given a state satisfying P , C will not produce a fault and, if it terminates, will deliver a state satisfying Q . More mathematically: $s, h \models P \wedge (s, h) \llbracket C \rrbracket \sigma \implies \sigma \neq \text{fault} \wedge \sigma \models Q$

This interpretation guarantees that C can only access heap which is guaranteed to exist by P . For, if C were to alter heap outside of an assertion P , then it would fault when that heap was deleted, and that would falsify $[P] C [Q]$.

4 Verification Condition Generation

Smallfoot chops an annotated program into Hoare triples for certain symbolic instructions, that are then decided using the symbolic execution mechanism of [5]. Execution reduces these triples to entailments $P \vdash Q$. These entailments are usually called verification conditions; we will use the same terminology for the output of the chopping phase, before the execution phase.

4.1 Verification Conditions

A verification condition is a triple $[P] SI [Q]$ where SI is a ‘‘symbolic instruction’’:

$$SI ::= \epsilon \mid S \mid [P] \text{jsr}_{\bar{x}} [Q] \mid \text{if } B \text{ then } SI \text{ else } SI \mid SI ; SI$$

A symbolic instruction is a piece of loop-free sequential code where all procedure calls have been instantiated to `jsr` instructions of the form $[P] \text{jsr}_{\vec{x}} [Q]$. This form plays a central role in Smallfoot. We use it not only to handle procedure calls, but also for concurrency and for entry to and exit from a critical region.

Semantically, $[P] \text{jsr}_{\vec{x}} [Q]$ is a “generic command” in the sense of [38]. It is the greatest relation satisfying the pre- and post-condition, and subject to the constraint that only the variables in \vec{x} are modified. An adaptation of generic commands which requires the relation to agree on local and pull out/slot in interpretations of triples, can be found in [33].

Overall, what the execution mechanism does for $[P] SI [Q]$ is start with P and run over statements in SI generating postconditions. For each postcondition P' thus obtained, it checks an entailment $P' \vdash Q$ using a terminating proof theory.

We will not give a detailed description of the symbolic execution mechanism, referring to [5] for the details. (We remark that the presentation there does not include conditional assertions if B then P else Q , but these are easily dealt with.) Instead, we will describe how the mechanism works in a particular case, in the else branch of the `disp_tree` program.

When we take that branch we have to establish a triple

$$[p \neq \text{nil} \wedge \text{tree}(p)] C [\text{emp}]$$

where C is the command in the else branch, with procedure calls instantiated to `jsr` instructions. Applying the tree unroll rule yields

$$[p \neq \text{nil} \wedge (p \mapsto l: i', r: j') * \text{tree}(i') * \text{tree}(j')] C [\text{emp}]$$

for fresh variables i' and j' . After the first two assignment statements in C we are left with:

$$\begin{aligned} & [p \neq \text{nil} \wedge (p \mapsto l: i, r: j) * \text{tree}(i) * \text{tree}(j)] \\ & ([\text{tree}(i)] \text{jsr} [\text{emp}]); ([\text{tree}(j)] \text{jsr} [\text{emp}]); \text{dispose}(p) [\text{emp}] \end{aligned}$$

To apply $[\text{tree}(i)] \text{jsr} [\text{emp}]$ we have to find a frame axiom, using the frame rule from earlier, and it is just $(p \mapsto l: i, r: j) * \text{tree}(j)$. Similarly, in the next step we obtain $p \mapsto l: i, r: j$ as the frame axiom, and finally we dispose p . (Frame inference is not always so easy; for example, CCR examples later require a certain amount of logical reasoning beyond pattern matching.) This leaves us with an easy entailment:

$$p \neq \text{nil} \wedge \text{emp} \vdash \text{emp}$$

4.2 VCGen

For each procedure declaration $f(\vec{p}; \vec{v}) [P] C [Q]$ we generate a set of verification conditions $\text{vcg}(f, [P] C [Q])$. The formal definition can be found in [4], and here we illustrate how it applies to the `par_disp_tree` and heaplet transfer examples presented in Sections 2.2 and 2.3.

Recall the specification of `par_disp_tree`: $[tree(p)] \text{par_disp_tree}(p) [\text{emp}]$. So for a call `par_disp_tree(i)`, *vcg* considers a single generic command:

$$[tree(i)] \text{jsr} [\text{emp}]$$

which indicates that the net effect of calling `par_disp_tree` on *i* is to consume a `tree` starting from *i*, produce no heap, and modify no (nonlocal) variables in the process. Using this straight-line command, and the similar one for the call `par_disp_tree(j)`, the net effect of the recursive parallel function calls `par_disp_tree(i) || par_disp_tree(j)` is to consume trees starting at *i* and *j*, produce no heap, and modify no variables. This is the core of the verification condition of `par_disp_tree`, and is expressed by the straight-line command:

$$[tree(i) * tree(j)] \text{jsr} [\text{emp}]$$

With this, the whole body of `par_disp_tree` is expressed by a conditional command, and so `par_disp_tree`'s single VC is obtained by tacking on the pre- and post-conditions:

$$\begin{array}{l} [tree(p)] \\ \text{if } p=0 \text{ then } \epsilon \text{ else } i := p \rightarrow l; j := p \rightarrow r; ([tree(i) * tree(j)] \text{jsr} [\text{emp}]); \text{dispose}(p) \\ [\text{emp}] \end{array}$$

This VC is then discharged by symbolic execution, which propagates the precondition forward through the command and then checks (for each branch of the execution) that the computed postcondition entails the specified one.

For the heaplet transfer example, the `init` procedure must establish the resource invariant from precondition `emp`, yielding VC:

$$[\text{emp}] \text{jsr} [\text{if } c=\text{nil} \text{ then } \text{emp} \text{ else } c \mapsto]$$

For brevity, if we inline `put` and `get` in `putter` and `getter`:

<pre> putter() [emp] { local x; x := new(); with buf when(c=nil) { c := x; } putter(); } [emp]</pre>	<pre> getter() [emp] { local y; with buf when(c≠nil) { y := c; c := nil; } dispose(y); getter(); } [emp]</pre>
------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------

The crux of the VCs of these functions is the straight-line command which expresses the CCR commands. For `getter` this is:

$$\begin{array}{l} [\text{emp}] \text{jsr} [(\text{if } c=\text{nil} \text{ then } \text{emp} \text{ else } c \mapsto) \wedge c \neq \text{nil}]; \\ y := c; c := \text{nil} \\ [\text{if } c=\text{nil} \text{ then } \text{emp} \text{ else } c \mapsto] \text{jsr}_c [\text{emp}] \end{array}$$

The generic commands for CCR entry and exit act as resource transformers. Recalling that the resource invariant for *buf* is $(\text{if } c=\text{nil} \text{ then emp else } c\rightarrow)$, the initial generic command expresses that upon entry into the CCR, the guard holds and resource invariant is made available to the body. Notice how the invariant is obtained starting from **emp** as a precondition, “materializing” inside the CCR as it were. Then the body runs, and the final generic command expresses that the body must reestablish the resource invariant prior to exiting the CCR.

The CCR in **putter** works similarly, but illustrates resource transfer on exit:

$$\begin{aligned} & [\text{emp}] \text{jsr } [(\text{if } c=\text{nil} \text{ then emp else } c\rightarrow) \wedge c=\text{nil}]; \\ & c:=x \\ & [\text{if } c=\text{nil} \text{ then emp else } c\rightarrow] \text{jsr}_c [\text{emp}] \end{aligned}$$

The use of **emp** in the postcondition, considering that $x \neq \text{nil}$ since x will have just been allocated, effectively deletes the invariant $c\rightarrow$ from consideration, and the cell pointed-to by c will not be accessible to the code following the CCR.

The VCs for **putter** and **getter** are then:

$\begin{aligned} & [\text{emp}] \\ & x:=\text{new}(); \\ & [\text{emp}] \\ & \text{jsr} \\ & [(\text{if } c=\text{nil} \text{ then emp else } c\rightarrow) \wedge c=\text{nil}]; \\ & c:=x; \\ & [\text{if } c=\text{nil} \text{ then emp else } c\rightarrow] \text{jsr}_c [\text{emp}]; \\ & [\text{emp}] \text{jsr} [\text{emp}] \\ & [\text{emp}] \end{aligned}$	$\begin{aligned} & [\text{emp}] \\ & [\text{emp}] \\ & \text{jsr} \\ & [(\text{if } c=\text{nil} \text{ then emp else } c\rightarrow) \wedge c \neq \text{nil}]; \\ & y:=c; \\ & c:=\text{nil}; \\ & [\text{if } c=\text{nil} \text{ then emp else } c\rightarrow] \text{jsr}_c [\text{emp}]; \\ & \text{dispose}(y); \\ & [\text{emp}] \text{jsr} [\text{emp}] \\ & [\text{emp}] \end{aligned}$
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Note that, as usual, when verifying a recursive procedure, the procedure’s specification is assumed. Here, this means that each recursive call is replaced by a generic command with the procedure’s pre- and post-conditions.

The main command is then a parallel function call:

$$\text{putter}(); \quad || \quad \text{getter};$$

which gives the additional verification condition:

$$[\text{emp}] ([\text{emp}] \text{jsr} [\text{emp}]) [\text{emp}]$$

Note that in both of these examples, no analysis of potential interleavings of the executions of parallel commands is needed. Given the resource invariants, the concurrent separation logic treatment of CCRs allows us to just verify a few triples for simple sequential commands.

5 Further Examples

5.1 More on Trees

The specification of `disp_tree` does not use $*$, even though the proof does. An example that uses $*$ in its spec is:

```

copy_tree(q;p) [tree(p)] {
  local i,j,i',j';
  if (p = nil) { q := p; }
  else {
    i := p→l; j := p→r;
    copy_tree(i';i); copy_tree(j';j);
    q := new(); q→l := i'; q→r := j'; }
} [tree(q) * tree(p)]

```

The tree predicate that we use is not sensitive to the contents of the tree, only the fact that it is a tree. So, if in `copy_tree` the final two steps were

$$q \rightarrow l := j'; \quad q \rightarrow r := i';$$

then we would actually have an algorithm for rotating the tree, though it would satisfy the same spec. If, on the other hand, we mistakenly point back into the old tree

$$q \rightarrow l := i; \quad q \rightarrow r := j;$$

then an error is reported; we do not have separation on termination.

The tree predicate that we have used here is one that satisfies

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

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, the $*$ between the subtrees ensures that there is no sharing (it is not a DAG), and the use of `emp` in the base case ensures that there are no cells in a memory satisfying `tree(E)` other than those in the tree. The fact that the specification does not mention any data field is what makes this a shape specification, insensitive to the particular data.

This definition of `tree(E)` is not something that the user of Smallfoot sees; it is outside the fragment used by the tool (it has a quantifier). Reasoning inside the tool essentially uses rolling and unrolling of this definition. For instance, the proof step where we entered the `else` branch uses an entailment

$$p \neq \text{nil} \wedge \text{tree}(p) \vdash \exists x, y. (p \mapsto l: x, r: y) * \text{tree}(x) * \text{tree}(y)$$

together with stripping the existential (generating fresh variables) when the right-hand side is subsequently used as a precondition.

5.2 Linked Lists

We now give an example, using lists, that cannot be handled using simple (un)rolling of an inductive definition. We work with linked lists that use field `tl` for the next element. The predicate for linked-list segments is the least satisfying the following specification.

$$\text{ls}(E, F) \iff (E = F \wedge \text{emp}) \vee (E \neq F \wedge \exists y. E \mapsto \text{tl}: y * \text{ls}(y, F))$$

A complete linked list is one that satisfies $\text{ls}(E, \text{nil})$.

Consider the following Smallfoot program, where the pre and post use complete lists only, but the loop invariant requires a genuine segment $\text{ls}(x, t)$. (One would use genuine segments in pres and posts for, e.g., queues.):

```

append_list(x; y) [ls(x, nil) * ls(y, nil)] {
  if (x = nil) { x := y; }
  else {
    t := x; u := t→tl;
    while (u ≠ nil) [ls(x, t) * t→tl: u * ls(u, nil)] {
      t := u; u := t→tl; }
    t→tl := y; }
} [ls(x, nil)]

```

The most subtle part of reasoning in this example comes in the last step, which involves a triple

$$[\text{ls}(x, t) * t \rightarrow tl: \text{nil} * \text{ls}(y, \text{nil})] \quad t \rightarrow tl := y; \quad [\text{ls}(x, \text{nil})]$$

We use a symbolic execution axiom

$$[A * x \mapsto f: y] \quad x \mapsto f := z \quad [A * x \mapsto f: z]$$

to alter the precondition in-place, and then we use the rule of consequence with the entailment

$$\text{ls}(x, t) * t \rightarrow tl: y * \text{ls}(y, \text{nil}) \vdash \text{ls}(x, \text{nil})$$

of the postcondition. This entailment does not itself follow from simple unrolling of the definition of list segments, but is proven in the proof theory used within Smallfoot by applying the inductive definition to conclude $\text{ls}(t, \text{nil})$ from $t \rightarrow tl: y * \text{ls}(y, \text{nil})$, and then applying a rule that encodes the axiom

$$\text{ls}(E_1, E_2) * \text{ls}(E_2, \text{nil}) \vdash \text{ls}(E_1, \text{nil})$$

It is this axiom that does not follow at once from list rolling and unrolling; in the metatheory it would require a proof by induction.

Generally, for each hardwired inductive predicate Smallfoot uses a collection of such rules that are consequences of induction, but that can be formulated in a way that does not require enumeration of inductive hypotheses. The proof theory we have obtained in this manner is complete as well as terminating for entailments involving lists and trees [5].

This ability to prove inductive properties is one of the characteristics which sets this approach apart from Alias Types [39] and its descendants. Alias Types includes coercions to roll and unroll inductive types, but (as far as we understand) consequences of induction must be witnessed by loops.

A final comment on this example. In the loop invariant we did not include a $*$ -conjunct $\text{ls}(y, \text{nil})$, which would indicate that the loop preserves the listness of y . The reason we did not include this is that y 's list is outside the footprint of the loop; Smallfoot discovers it as a frame axiom.

5.3 Information Hiding

The following describes a toy memory manager, which maintains binary cons cells in a free list. When the list is empty, the `alloc(x;)` operation calls `new` in a way similar to how `malloc()` might call a system routine `sbrk()` to request additional memory.

```
resource mm (f) [ls(f, nil)]

init() { f := nil; }

alloc(x;) [emp] {
  with mm when(true) {
    if(f = nil) { x := new(); } else { x := f; f := x→tl; } }
} [x→]

dealloc(y) [y→] { with mm when(true) { y→tl := f; f := y; } } [emp]
```

The use of CCRs provides mutual exclusion, so that several calls to `alloc` or `dealloc` in different process will not interfere. The real point of the example, though, is information hiding. Because of the modularity of the CCR rule, the interface specifications for `alloc` and `dealloc` do not mention the free list at all. Furthermore, the specification of `dealloc` forces permission to access a deallocated cell to be given up, and this is essential to prevent incorrect usage.

For example, the little main program

```
main() { alloc(z;); dealloc(z); z→tl := z; }
```

is flagged as an error by Smallfoot, because the precondition to `z→tl := z` will be `emp`; we must know that `z` points to something to do a dereference, this program would tie a cycle in the free list.

However, the reason that this program is ruled out is not just because the invariant it violated, it is because the cell `z` (now in the free list) cannot be touched at all after `dealloc(z)`. For example, if we were to replace `z→tl := z` by `z→tl := nil` then the free list would not be corrupted in the global state, but the example still would not pass Smallfoot; it breaks abstraction by dereferencing a cross-boundary pointer, into the free list abstraction.

The effect of this information hiding can be seen more strongly by replacing the occurrences of `new` and `dispose` in the pointer-transferring buffer with calls to the homegrown memory manager.

```
putter() { alloc(x;); put(x); putter(); }

getter() { get(y;); /* use y */ dealloc(y); getter(); }
```

If we replace the `putter` and `getter` procedures from Section 2.3 with these, include joint initialization of the two resources

```
init() { f := nil; c := nil; }
```

and leave everything else the same, then the code verifies. If we did not use the CCR rule to hide resource invariants, we would have to “thread” the free list

through the buffer code, forcing us to alter the specifications of `put` and `get` by including $ls(f, nil)$ in their preconditions and postconditions.

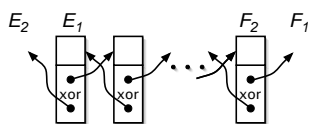
5.4 XOR-deqs

For our final example we consider DEQs – double-ended queues – implemented using an xor-linked list. Recall that an xor-linked list is a compact representation of doubly-linked lists where, instead of using separate fields to store previous and next nodes, their bitwise exclusive or is stored in one field [23]. Besides their entertainment value, using xor lists here demonstrates how Smallfoot does not depend on reachability. The fact that the separation logic triple $[P] C [Q]$ asserts that execution of C in states described by P will not access any other locations (except possibly locations newly allocated by C) does not depend on whether other such locations are reachable or not. We will also allow concurrent access to the DEQ, though that aspect is of secondary importance for the example.

The following predicate describes xor-linked list segments:

$$\begin{aligned} \text{xlseg}(E_1, F_1, E_2, F_2) \\ \stackrel{\text{def}}{\text{iff}} (E_1 = F_1 \wedge E_2 = F_2 \wedge \text{emp}) \\ \vee (E_1 \neq F_1 \wedge E_2 \neq F_2 \wedge \exists x. (E_1 \mapsto l: (E_2 \text{ xor } x)) * \text{xlseg}(x, F_1, E_1, F_2)) \end{aligned}$$

In reading this definition it helps to think of a picture:



The basic idea is that a resource will own the DEQ represented as an xor list, while processes accessing the two ends will hold dummy nodes *back* and *front* which will be used for putting elements in the DEQ. Operations for getting from the DEQ will release nodes into the calling processes. The resource declaration and invariant are as follows; additionally, an initialization is needed (code omitted) which sets up the pictured postcondition.

```
resource xdeq(n, p) [front=p ∧ back=n ∧ xlseg(f, n, p, b)]
init() { ... } [(front→l:prev xor f) * (back→l:next xor b)]
```

In this invariant it helps to consider the picture above: heaps cells corresponding to the nodes n and p (and hence *front* and *back*) are not held in the DEQ, but rather are pointers into the processes that hold them as dummy nodes.

There are four procedures for accessing the DEQ: in Table 1 we show the code for putting on the back and getting from the front, and the specifications only for the other two (their code is similar).

What the `getf` procedure does is dispose the dummy node *front* it currently has, replacing it with the first node f that is in the DEQ. This is done by an

Table 1

xor-linked DEQ accessors

```

getf(x;) [front→l:prev xor f] {
  local t, old_f;
  t := front→l;
  old_f := prev xor t;
  dispose(front);
  prev := front;
  /* split new dummy link
   off front */
  with xdeq when(old_f≠n) {
    t := f→l;
    f := t xor p;
    p := old_f;
    front := p;
  }
  x := front→d;
} [front→l:prev xor f]

getb(x;) [back→l:next xor b] {
  ...
} [back→l:next xor b]

putb(x) [back→l:next xor b] {
  local t, new_n, old_p;
  /* allocate new dummy link */
  new_n := new();
  new_n→l := next xor back;
  /* store datum in previous dummy,
   link to new dummy */
  back→d := x;
  t := back→l;
  old_b := t xor next;
  back→l := new_n xor old_b;
  /* move previous dummy link
   into DEQ */
  with xdeq when(p≠back) {
    b := back;
    n := new_n;
    back := n;
  }
} [back→l:next xor b]

putf(x) [front→l:prev xor f] {
  ...
} [front→l:prev xor f]

```

ownership transfer, within a critical region, similar to what was done in the pointer-transferring buffer in Section 2.3. Note that, although $front \rightarrow l: prev \text{ xor } f$ is true at the beginning and end of the procedure, it is false at intermediate points. Similarly, the `putb` procedure stores a new item in the d field of its dummy node $back$, and then effects an ownership transfer where this node gets swallowed into the DEQ data structure. The crucial point is that the accesses $x := front \rightarrow d$ and $back \rightarrow d := x$ of the data fields occur outside of critical sections. It is this that allows these accesses to be done in parallel.

[Aside: There is one limitation of Smallfoot that this example shows: in the `putb` procedure we include a test ($p \neq back$) which happens to be true in any execution. This condition is an additional annotation that Smallfoot needs to verify the code. The difficulty is that two processes may be allocating nodes concurrently, and the allocated nodes will indeed be different, but our current assertions do not allow us to say so, locally. We say “current” because if we change the memory model to allow “existence permissions” [7, 8] then it is possible to do away with the extra annotation in a by-hand proof; we have not, though, incorporated existence permissions into Smallfoot as of yet.]

To show these procedures working, we set up two parallel processes `procf` and `procb` which nondeterministically choose whether to do a put or get operation on

the two ends of the DEQ. (The `nondet` keyword was not in the formal grammar for Smallfoot before, but it is in the tool and is implemented by talking both branches of a conditional in symbolic execution.)

```

procf(x) [front→l: prev xor f] {
  if(nondet) {
    getf(x); /* use x */ }
  else {
    /* produce an x */ putf(x); }
  procf(x);
} [false]

procb(x) [back→l: next xor b] {
  if(nondet) {
    getb(x); /* use x */ }
  else {
    /* produce an x */ putb(x); }
  procb(x);
} [false]

main() procf(42) || procb(13);

```

Smallfoot verifies the resulting program using a terminating proof theory for facts about xor lists. It involves basic identities for xor, together with adaptations of the rules in [5] for list segments. Again, this example could not be verified without consequences of induction that go beyond rolling and unrolling of an inductive definition, and Smallfoot uses several such for xor lists, akin to the axiom described in Section 5.2.

This is a variant of classic algorithms which allow concurrent access to two ends of a queue. As usual, we could allow multiple processes at each end of the queue by using mutexes to rule out concurrent accesses from the same end.

6 Conclusions and Related Work

Before discussing related work we mention some of Smallfoot’s limitations.

First, even when a program’s preconditions and postconditions can be expressed using Smallfoot assertions, we will not be able to verify it if its (loop and resource) invariants cannot be expressed. An example of this is Parkinson and Bornat’s proof [34] of the non-blocking stack of Michael [26]. (Parkinson has verified a different non-blocking algorithm which is included amongst the examples on our web pages, but we are unable to express the invariant for Michael’s algorithm.)

Incidentally, although Brookes has shown that concurrent separation logic rules out races [11], this should not be taken to mean that it cannot be used on programs that are normally considered racy. Generally, one can use little CCRs to explicitly notate statements that are considered atomic, or one could use some other notation (e.g., “atomic”) with the same proof methodology, and that is what Parkinson and Bornat have done in [34].

Second, Smallfoot uses a strict separation model, which does not allow sharing of read access. As a consequence it cannot handle, e.g., a readers and writers program, which is proven in [8] using a less strict “counting permissions” model of separation logic. Adding permission accounting is on our to-do list.

Third, it would be straightforward to include inductive definitions, if we were content to just roll and unroll them. However, then very many interesting

programs would not verify. Several examples in this paper involving linked lists required properties to be proven at both ends, and these would not be verifiable using rolling and unrolling alone. A direction for future research is to find a class of inductive definitions and to classify consequences of induction that can be included in a terminating proof theory.

The most closely related works to Smallfoot are the Pointer Assertion Logic Engine [28] and Alias Types [39] and its relatives (e.g. [13, 9]). PALE is stronger than Smallfoot in the range of predicates it considers, based on graph types. The state of development of Smallfoot is more directly comparable to the first version of PALE [20], which was for linked lists only, and we hope to encompass some graph structures in the future. Conversely, PALE does not check frame conditions on recursive calls, and this (intentionally) leads to unsoundness, whereas the treatment of framing is a focus of Smallfoot. Also, PALE does not deal with concurrency. Early on in the Smallfoot development we considered whether we could translate a fragment of separation logic into the fragment of monadic second-order logic that PALE is based on. For some specific assertions it is possible but we were unable to find a general scheme. The decidability of fragments of monadic second-order logic is brittle, and can be broken by adding features. Most importantly, we were unable to see how to give a compositional interpretation of $*$.

With regard to Alias Types, there are many similarities. Most importantly, both approaches use a substructural logic or type theory for heaps. We believe it is fair to say that the annotation burden in Smallfoot is considerably less than in Alias Types, owing mainly to inference of frame axioms. Alias Types were aimed at intermediate languages, so that is not a criticism of them. Another difference is that Alias Types use a range of inductive predicates, while we only use several specific predicates. However, our proof theory uses strong and sometimes complete inductive properties, such as are needed when working at both ends of a linked list.

The shape analysis of Sagiv, Reps and Wilhelm [37] provides a powerful verification technique for heaps. The biggest problem with the approach is that it is non-modular, in that an update to a single abstract heap cell can necessitate changing the whole abstract heap (some steps to build in modularity have been taken in [36]). We considered whether we could use ideas from shape analysis within a single procedure, and leverage $*$'s modularity for interprocedural and concurrent analysis. Again, we had great difficulty dealing with $*$ compositionally, and further difficulties with the `dispose` instruction. But investigations continue; if this direction worked out it would give us access to a much wider range of abstractions (using “canonical abstraction” [37]).

We are often asked: why did you not just give a deep (semantic) embedding of separation logic in a predicate logic, and then use a general theorem prover, instead of constructing your own proof method? The short answer is that the deep embedding leads to nested quantifiers in the interpretation of $*$, and this is an impediment to automation; attempts so far along these lines have proven to be highly non-automatic. Of course it would be valuable to construct a deep

embedding and develop a range of tactics and make use of general purpose provers, but that is for separate work.

Work on ESC and Spec# has resulted in important advances on modular heap verification [16, 3]. Ideas of ownership and inclusion have been used to classify objects, and give a way of avoiding frame axioms, intuitively related to work on ownership types and semantics [12, 1]. Methods based on fixed ownership structures have been described (e.g., [25, 14]), but fixed structures are inflexible, e.g., having difficulty dealing with ownership transfer examples (like our pointer-transferring buffer or memory manager), except possibly under draconian restrictions (such as unique-pointer restrictions). A recent emphasis has been on using ownership assertions that refer to auxiliary fields that may be altered, and this leads to added flexibility [2, 24], including transfer. New schemes are being invented to extend the basic idea, such as a “friends” concept that lets invariants reach across hierarchical ownership domain [30]. We refer to David Naumann’s survey paper for a fuller account of and further references to research in this area [29].

In contrast, separation logic does not require a hierarchical ownership structure to ensure locality or encapsulation. Assertions just describe heaplets, portions of state, and an assertion encapsulates all of the state that a command is allowed to change. Still, there appear to be some similarities between the ownership assertion approaches and the reasons for why separation logic works modularly [41, 33]; there seem to be, in particular, remarkably similar intuitions underlying a recent ownership-invariant system for concurrency [19] and concurrent separation logic [31]. A careful comparison of their models could be worthwhile.

Modular reasoning about concurrent programs has also received much attention, often based on the fundamental work of Jones, Misra and Chandy [21, 27]. Our remarks at the end of Section 2.2 apply also in any comparison between Smallfoot and tools based on rely/guarantee (e.g. [15]). Our remarks should not be taken to be an ultimate argument for separation logic over rely/guarantee, and it would be interesting to attempt to marry their strong points (easy treatment of independence, powerful treatment of dependence). We should add that the criticisms we made echo comments made by Jones himself [22].

Smallfoot is written in OCaml, and all of the examples in this paper verified in a few milliseconds on an ordinary laptop. We have not included a timing table or other experimental results, because for the small examples we have considered the interpretation of such results would be questionable, except that if the verifications had taken minutes or hours and not milliseconds then that would have been a negative indication. The source code for the current version of Smallfoot (v0.1), together with the examples from this paper and several others, is available for download from the address given in reference [4].

ACKNOWLEDGMENTS. Thanks to Matthew Parkinson, who wrote lots of little programs that tested the tool and pointed out a number of bugs. All three authors were partially supported by the EPSRC. During Berdine’s stay at Carnegie Mellon University during 2003, his research was sponsored by National Science Foundation Grant CCR-0204242.

References

1. A. Banerjee and D.A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005. Preliminary version in POPL’02.
2. M. Barnett, R. DeLine, M. Fahndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
3. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS’04 post-proceedings*, 2004.
4. J. Berdine, C. Calcagno, and P.W. O’Hearn. Verification condition generation and variable conditions in Smallfoot. Available from <http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/index.html>.
5. J. Berdine, C. Calcagno, and P.W. O’Hearn. Symbolic execution with separation logic. In *3rd APLAS*, pages 52–68, 2005.
6. L. Birkedal, N. Torp-Smith, and J.C. Reynolds. Local reasoning about a copying garbage collector. In *31st POPL*, pages 220–231, 2004.
7. R. Bornat, C. Calcagno, and P. O’Hearn. Local reasoning, separation, and aliasing. Presented at *2nd SPACE Workshop*, 2004.
8. R. Bornat, C. Calcagno, P. O’Hearn, and M. Parkinson. Permission accounting in separation logic. *32nd POPL*, 59–70, 2005.
9. J. Boyland. Checking interference with fractional permissions. In *10th SAS*, pages 55–72, 2003.
10. P. Brinch-Hansen, editor. *The Origin of Concurrent Programming*. Springer-Verlag, 2002.
11. S.D. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, to appear. Preliminary version in *CONCUR’04*, 2006.
12. D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *15th ECOOP*, pages 53–76, 2001.
13. R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *8th PLDI*, pages 59–69, 2001.
14. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 2006.
15. C. Flanagan, S.N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *11th ESOP*, pages 262–277, 2002.
16. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *9th PLDI*, pages 234 - 245, 2002.
17. C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on the Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer-Verlag, 1971.
18. C.A.R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, Acad. Press, pages 61–71. Reprinted in [10], 1972.
19. B. Jacobs, K.R.M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *3rd SEFM*, 2005.
20. J. Jenson, M. Jorgensen, N. Klarkund, and M. Schwartzback. Automatic verification of pointer programs using monadic second-order logic. In *4th PLDI*, pages 225–236, 1997.
21. C.B. Jones. Specification and design of (parallel) programs. *IFIP Conf.*, 1983.
22. C.B. Jones. Wanted: A compositional approach to concurrency. In A. McIver and C. Morgan, editors, *Programming Methodology*, pages 1–15, 2003. Springer-Verlag.

23. D.E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison Wesley, 2nd edition, 1973.
24. K.R.M. Leino and P. Müller. Object invariants in dynamic contexts. In *18th ECOOP*, pages 491-516, 2004.
25. K.R.M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *9th PLDI*, pages 246 - 257, 2002.
26. M.M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 15(6):491–504, 2004.
27. J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
28. A. Möller and M.I. Schwartzbach. The pointer assertion logic engine. In *8th PLDI*, pages 221-231, 2001.
29. D.A. Naumann. Assertion-based encapsulation, invariants and simulations. In *3rd FMCO*, pages 251-273, 2005.
30. D.A. Naumann and M. Barnett. Friends need a bit more: Maintaining invariants over shared state. In *7th MPC*, pages 54-84, 2004.
31. P.W. O’Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, to appear. Preliminary version in *CONCUR’04*, 2006.
32. P.W. O’Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*. pages 1-19, 2001.
33. P.W. O’Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. In *31st POPL*, pages 268-280, 2004.
34. M. Parkinson and R. Bornat. Exploiting linearisability in program logic. Draft paper, 2005.
35. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pages 55-74.
36. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *32nd POPL*, pages 296–309, 2005.
37. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
38. J. Schwarz. Generic commands—A tool for partial correctness formalisms. *The Computer Journal*, 20(2):151–155, 1977.
39. D. Walker and J.G. Morrisett. Alias types for recursive data structures. In *3rd Types in Compilation Workshop*, pages 177-206, 2001.
40. H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. Presented at *1st SPACE Workshop*, 2001.
41. H. Yang and P.W. O’Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, pages 402–416, 2002.