

Footprint Analysis: A Shape Analysis that Discovers Preconditions

Cristiano Calcagno¹, Dino Distefano², Peter W. O’Hearn², and
Hongseok Yang²

¹ Imperial College, London

² Queen Mary, University of London

Abstract. Existing shape analysis algorithms infer descriptions of data structures at program points, starting from a given precondition. We describe an analysis that does not require any preconditions. It works by attempting to infer a description of only the cells that might be accessed, following the footprint idea in separation logic. The analysis allows us to establish a true Hoare triple for a piece of code, independently of the context in which it occurs and without a whole-program analysis. We present experimental results for a range of typical list-processing algorithms, as well as for code fragments from a Windows device driver.

1 Introduction

Existing shape analysis engines (e.g., [25, 9, 15, 13, 4]) require a precondition to be supplied in order to run. Simply put, this means that they cannot be used automatically without either knowing the execution context (which might be an entire operating system, or even be unknown) or by manually supplying a precondition (which for complex code can be hard to determine). If, though, we could discover preconditions then, combined with a usual forwards-running shape analysis, we could automatically generate true Hoare triples for pieces of code independently of their context.

This paper defines *footprint analysis*, a shape analysis that is able to discover preconditions (as well as postconditions). Our results build on the work on shape analysis with separation logic [9]; the footprint analysis algorithm is itself parameterized by a standard shape analysis based on separation logic. In essence, we are leveraging the “footprint” idea of [21]. Separation logic gives us mechanisms whereby a specification can concentrate on only the cells accessed by a program, while allowing the specification to be used in wider contexts via the “frame rule”. For program analysis this suggests, when considering a code fragment in isolation, to try to discover assertions that describe the footprint, rather than the entire global state of the system. This is the key idea that makes our analysis viable: the entire global state can be enormous, or even unknown, where we can use much smaller assertions to talk about the footprint.

Footprint analysis runs forwards, updating the current heap when it can in the usual way of shape analysis. However, when a dereference to a potentially-

dangling pointer is encountered, that pointer is added into the “footprint assertion”, which describes the cells needed for the program to run safely. If we start the analysis with the empty heap as the initial footprint assertion then, ideally, it would find the collection of safe states, ones that do not lead to a dereference of a dangling pointer or other memory fault.

We say ideally here because there is a complication. In order to stop the footprint assertion from growing forever it is periodically abstracted. The abstraction we use is an overapproximation and, usually in shape analysis, this leads to incompleteness while maintaining soundness. But, abstracting the footprint assertion is tantamount to *weakening a precondition*, and so for us is a potentially unsound step. As a result, we also use a post-analysis phase, where we run a standard forwards shape analysis to filter out the unsafe preconditions that have been discovered. For each of the safe preconditions, we also generate a corresponding postcondition.

The source of this complication is, though, also a boon. In shape domains it can be the case that a reasonably general assertion can be obtained from a specific concrete heap using the domain’s abstraction function. For example, a linked list of length three is often abstracted as a linked list of unknown length. This nature of the shape domains is what lets footprint analysis often find a reasonably general precondition, which is synthesized from concrete assertions generated when we encounter potential memory faults.

We show by experimental results that footprint analysis is indeed able to discover non-trivial preconditions, in a number of cases resembling the precondition that we would normally write by hand. Intuitively, the algorithm works well because pointer programs are often insensitive to the abstractions we use, and so the step for filtering out unsound preconditions often does nothing. A limitation of the paper is that we do not have a thorough theoretical explanation to back this intuition up,³ so the method might be regarded as having a heuristic character. We felt it reasonable to describe our discovery algorithm now because the results of the analysis are encouraging, and the algorithm itself employs the footprint idea in a novel way. Also, there are several potential further applications of having in place an analysis that discovers preconditions, which we describe at the end of the paper. We hope that further theoretical understanding of our method will be forthcoming in the future.

Context and Further Discussion. For precondition discovery one of the first things that come to mind is to use an underapproximating backwards analysis. While possible in principle, we have found it difficult to obtain precise and efficient backwards analyses for shape domains. As far as we are aware the problem of finding a useful backwards-running shape analysis is open.

Footprint analysis can be seen as an instance of the general idea of relational program analysis [8]. The purpose of a relational analysis is to compute an

³ Mooly Sagiv has suggested starting from a \top value and homing in on the needed states using a greatest fixed-point computation, as in [27]. We have not been able to make that approach work, and it does not describe what our analysis is doing, but it and other approaches are worth exploring.

```

1: while (c!=NULL) {
2:     t=c;
3:     c=c->t1;
4:     free(t);
5: }

```

Discovered Precondition: $c==c_ \wedge \text{lseg}(c_ , \text{NULL})$

Fig. 1. Program `delete_list`, and discovered precondition when run in start state $(c==c_ \wedge \text{emp}, \text{emp})$

overapproximation of the transition relation of a program. After the post-analysis check to filter out unsound preconditions, footprint analysis returns a set of true Hoare triples for a program, and from this it is easy to construct the relational overapproximation.

The shape analysis of [14] tracks relationships between input and output heaps. In the examples there, a precondition was typically supplied as input; for example, for in-place list reversal the input indicated an acyclic linked list. However, it might be possible to use a similar sort of idea to replace the separate preconditions and postconditions used in our algorithm, which might result in an improved precondition discovery method.

2 Basic Ideas

In this section we illustrate how our algorithm finds a precondition via a fixed-point calculation, using a simple example. The paper continues in the next section with the formal development.

The abstract states in the analysis consist of two assertions (H, F) , represented as separation logic formulae (see [22] for the basics of separation logic). H represents the currently known or allocated heap and F the cells that are needed (the footprint). As described above, the analysis runs forwards, adding pointers into the footprint assertion F when dereferences to potentially dangling pointers are encountered. In doing this care is needed in the treatment of variables, especially what we call *footprint variables*.

The algorithm is attempting to discover a precondition that describes “safe heaps”, ones that do not lead to a dereference of a dangling pointer or other memory fault when the program in question is run. We illustrate with a program that disposes all the elements in an acyclic linked list. Footprint analysis discovers the precondition pictured in Figure 1, which says that c points to a linked list segment terminating at `NULL`. This precondition describes just what is needed in order for the program not to dereference a dangling pointer during execution. We now outline how footprint analysis finds this assertion.

We begin symbolic execution with $c==c_ \wedge \text{emp}$ as the current heap and emp as the footprint. Note that $c==c_$ allows for a state where c (or any other location) is dangling. The current heap includes a footprint variable $c_$, and assertion emp

	Current Heap	Footprint Heap
First iteration		
pre:	$c \neq \text{NULL} \wedge c == c_ \wedge t == c_ \wedge \text{emp}$	emp
post:	$t \neq \text{NULL} \wedge c == c1_ \wedge t == c_ \wedge c_ \mapsto c1_$	$c_ \mapsto c1_$
Second Iteration		
pre:	$c \neq \text{NULL} \wedge c == c1_ \wedge t == c1_ \wedge \text{emp}$	$c_ \mapsto c1_$
post:	$t \neq \text{NULL} \wedge c == c2_ \wedge t == c1_ \wedge c1_ \mapsto c2_$	$c_ \mapsto c1_ * c1_ \mapsto c2_$
abs post:	$t \neq \text{NULL} \wedge c == c2_ \wedge t == c1_ \wedge c1_ \mapsto c2_$	$\text{lseg}(c_ , c2_)$
Third Iteration		
pre:	$c \neq \text{NULL} \wedge c == c2_ \wedge t == c2_ \wedge \text{emp}$	$\text{lseg}(c_ , c2_)$
post:	$t \neq \text{NULL} \wedge c == c3_ \wedge t == c2_ \wedge c2_ \mapsto c3_$	$\text{lseg}(c_ , c2_) * c2_ \mapsto c3_$
abs post:	$t \neq \text{NULL} \wedge c == c3_ \wedge t == c2_ \wedge c2_ \mapsto c3_$	$\text{lseg}(c_ , c3_)$

Fig. 2. Pre and Post States at line 3 during footprint analysis of `delete_list`

represents the empty heap. When execution enters the loop and gets to line 3, we will attempt a heap dereference to $c \rightarrow t1$ but where we do not know that c is allocated in the precondition. This is represented in the precondition for the first iteration in Figure 2. At this point the knowledge that c points to something is added to the footprint: we need that information in order for our program not to commit a memory fault. Also, though, in order to continue symbolic execution from this point, this knowledge is added to the allocated heap as well, as pictured in the postcondition for the first iteration in Figure 2. Notice that we express that c points to something in terms of the footprint variable $c_$. Because it is not a program variable, and so not changed by the program, this will enable us to percolate the footprint information back to the precondition.⁴

Now, the next statement in the program, line 4, removes the assertion $c_ \mapsto c1_$ from the current heap, using the knowledge that $t = c_$, but that assertion is left in the footprint. Then, when we execute the second iteration of the loop we again encounter a state where c is not allocated in the current heap. At this point we again add a pointer to the footprint and to the current heap: see the pre and post for the second iteration in Figure 2. The assertion in the footprint part uses the separating conjunction $*$, which requires that the conjuncts hold for separate parts of memory (and so here, denote distinct cells). Notice that the footprint variable $c1_$ was known to equal c in the precondition. Also, in the postcondition we generate another footprint variable, $c2_$.

So, after two iterations, we have found a linked list of length two in the footprint. But, this way of generating new footprint variables is a potential source of divergence in the analysis. In order to enable the fixed-point calculation to converge we abstract the footprint part of the assertion, as indicated in Figure

⁴ Notice also, though, that an additional footprint variable $c1_$ is added: the footprint variables resemble those typically used for seeding initial program states, but seeding does not cover all of their uses.

2, and the footprint now says that there is a list segment from c_1 to c_2 . This abstraction step has lost the information that the list is of length two, in that the assertion is satisfied by lists of length three, four, and so on.⁵

Continuing our narrative symbolic execution, the `free` statement will delete the assertion “ $c_1 \mapsto c_2$ ” from the current heap (but not the footprint), and when we go into the third iteration we will again try to dereference $c \rightarrow t_1$ when it is not known to be allocated from the current heap in the `free` command. We put a \mapsto assertion in the current and footprint parts again, and then abstract. Now, when we apply abstraction the assertion “ $c_2 \mapsto c_3$ ” is swallowed into the list segment. Except for the names of newly-generated footprint variables, the abstracted post we obtained in the second iteration is the same as in the third, and we view the newly-generated footprint variables as alpha-renameable. The reader can see the relevance to fixed-point convergence.

Finally, we can exit the loop by removing “ $c_2 \mapsto c_3$ ” from the current heap in the `free` command, and adding the negation of the loop conditional to the heap and footprint, and forgetting about t because it is a local variable. A bit of logic tells us that the footprint part is equivalent to $c_3 = \text{NULL} \wedge \text{lseg}(c_1, c_3)$, and when we add this to the initial precondition $c = c_1$ we obtain the overall precondition pictured in Figure 1.

3 Programming Language and Generic Analysis

In this section we define the programming language used in the formal part of the paper. We also set up a generic analysis, following the tradition of abstract interpretation [7], which will have the shape and footprint analyses described later as instances.

Programming Language. In the paper, we use a simple `while` language extended with heap operations:

$$\begin{aligned}
 E, F &::= x \mid f(E_1, \dots, E_n) \\
 b &::= E = F \mid E \neq F \\
 a[E] &::= [E] := F \mid \text{dispose}(E) \mid x := [E] \\
 a &::= x := E \mid x := \text{new}(E) \\
 c &::= a[E] \mid a \mid c_1; c_2 \mid \text{if } b \text{ } c_1 \text{ } c_2 \mid \text{while } b \text{ } c
 \end{aligned}$$

An expression E is either a variable or a heap-independent term $f(E_1, \dots, E_n)$. The language has two classes of atomic commands. $a[E]$ attempts to dereference cell E , updating it ($[E] := F$), disposing it ($\text{dispose}(E)$), or reading its content ($x := [E]$). The other atomic commands, denoted a , do not access existing cells.

⁵ This step of abstraction depends on which abstract domain we plug into our footprint analysis; several have appeared in the literature, and the footprint analysis does not depend on any one choice. In this example, we have assumed that the “`lseg`” predicate describes “possibly circular list segments”, which allows the abstraction step we have done. If circularity were outlawed in our abstract domain, as in the particular domain of [9], then one more loop iteration would be needed before abstraction could occur.

Generic Analysis. The analyses in this paper will use the topped powerset $\mathcal{P}^\top(S)$ of a set S ; i.e., the powerset with an additional greatest element. A set $X \in \mathcal{P}(S)$ represents a disjunction of its elements $x \in X$, and \top indicates that the analysis detected an error in a given program. When $D = \mathcal{P}^\top(S)$, we call S the underlying set of the abstract domain D .

Given function $t: S \rightarrow \mathcal{P}^\top(S')$ and partial or total function $f: S \rightarrow S'$, we can lift them to functions $t^\dagger, \mathcal{P}^\top(f): \mathcal{P}^\top(S) \rightarrow \mathcal{P}^\top(S')$ by

$$\begin{aligned} t^\dagger(X) &\stackrel{\text{def}}{=} \mathbf{if} (X = \top) \mathbf{then} \top \mathbf{else} \bigsqcup_{x \in X} t(x) \\ \mathcal{P}^\top(f)(X) &\stackrel{\text{def}}{=} \mathbf{if} (X = \top) \mathbf{then} \top \mathbf{else} \{f(x) \mid x \in X\}. \end{aligned}$$

The generic analysis framework consists of the following data.

- (1) A set S of abstract states, inducing the abstract domain $D = \mathcal{P}^\top(S)$, which forms a complete lattice $(D, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$.
- (2) For all boolean expressions b , atomic commands $a, a[E]$ and expressions E , the operators

$$\begin{aligned} \mathbf{rearr}(E): S &\rightarrow \mathcal{P}^\top(S[E]), & \mathbf{filter}(b): S &\rightarrow S, \\ \mathbf{exec}(a[E]): S[E] &\rightarrow S, & \mathbf{exec}(a): S &\rightarrow S, & \mathbf{abs}: S &\rightarrow S. \end{aligned}$$

Here $S[E]$ is a subset of S , and it consists of symbolic states where cell E is explicitly represented by a points-to fact $E \mapsto E'$.

This framework does not ask for transfer functions to be given directly, but rather asks for more refined ingredients, out of which transfer functions are usually made in shape analysis. $\mathbf{rearr}(E)$ typically takes a symbolic state and attempts to “concretize” cell E , making it a points-to fact of the form $E \mapsto E'$. When instantiating the generic analysis with the one in [9], this operation corresponds to unwinding an inductive definition, and when instantiating with [24] it is the materialization of a summary node. The abstraction map \mathbf{abs} simplifies states, as illustrated in the example in the previous section. In [9, 25] it is called canonicalization. $\mathbf{filter}(b)$ is used to filter states that do not satisfy boolean condition b , and $\mathbf{exec}(a[E])$ and $\mathbf{exec}(a)$ implement update (after rearrangement).

Given this data, abstract transfer functions of the primitive commands are:⁶

$$\begin{aligned} \llbracket b \rrbracket &\stackrel{\text{def}}{=} \mathcal{P}^\top(\mathbf{filter}(b)) \\ \llbracket a[E] \rrbracket &\stackrel{\text{def}}{=} (\mathcal{P}^\top(\mathbf{abs} \circ \mathbf{exec}(a[E])) \circ \mathbf{rearr}(E))^\dagger & \llbracket a \rrbracket &\stackrel{\text{def}}{=} \mathcal{P}^\top(\mathbf{abs} \circ \mathbf{exec}(a)). \end{aligned}$$

The execution of a command accessing E is done in three steps: first the cell E is exposed by $\mathbf{rearr}(E)$, then the state is updated according to the semantics of the command $a[E]$ by $\mathbf{exec}(a[E])$ and finally the resulting state is abstracted by \mathbf{abs} . The execution of a command a that does not access the heap does not

⁶ Our analysis specification presumes that abstraction is applied after every transfer function, but it is also possible to instead take it out of the transfer functions and apply only often enough to allow the loop computations to converge.

involve the rearrangement phase. The reader is referred to [9] for an extensive treatment of transfer functions defined in terms of `rearr`, `exec` and `abs`.

We may then define monotone functions $\llbracket c \rrbracket: D \rightarrow D$ for each command c in the usual way of abstract interpretation.⁷

$$\begin{aligned} \llbracket c_1; c_2 \rrbracket &= \llbracket c_2 \rrbracket \circ \llbracket c_1 \rrbracket & \llbracket \text{if } b \text{ } c_1 \text{ } c_2 \rrbracket(d) &= (\llbracket c_1 \rrbracket \circ \llbracket b \rrbracket)(d) \sqcup (\llbracket c_2 \rrbracket \circ \llbracket \neg b \rrbracket)(d) \\ \llbracket \text{while } b \text{ } c \rrbracket(d) &= \llbracket \neg b \rrbracket(\llbracket \text{fix } \lambda d'. d \sqcup (\llbracket c \rrbracket \circ \llbracket b \rrbracket)(d') \end{aligned}$$

4 Underlying Shape Analysis based on Separation Logic

We assume that we are given three disjoint countable sets of variables:

- Vars for program variables x, y ;
- PVars for primed variables x', y' ;
- FVars for footprint variables \bar{x}, \bar{y} .

Let **Locs** and **Vals** be countable infinite sets of locations and values, respectively, such that $\text{Locs} \subseteq \text{Vals}$. When V is set to be the union of **Vars**, **PVars** and **FVars**, our concrete storage model is given by:

$$\text{Stacks} \stackrel{\text{def}}{=} V \rightarrow \text{Vals} \quad \text{Heaps} \stackrel{\text{def}}{=} \text{Locs} \rightarrow_{\text{fin}} \text{Vals} \quad \text{States} \stackrel{\text{def}}{=} \text{Stacks} \times \text{Heaps}.$$

Each state consists of stack and heap components. The stack component s records the values of program, primed and footprint variables, and the heap component h specifies the identities and contents of allocated cells. Note that this model can allow data structures of complex shape, because a pair of addresses can be a value so a cell can have two outgoing pointers.

The analysis described in this paper uses separation logic assertions (called *symbolic heaps*) to represent abstract states. Symbolic heaps H are given by the following grammar:

$$\begin{aligned} E, F &::= \text{nil} \mid x \mid x' \mid \bar{x} \mid \dots \\ \Pi &::= \text{true} \mid E = E \mid E \neq E \mid \Pi \wedge \Pi \mid \dots \\ \Sigma &::= \text{true} \mid \text{emp} \mid E \mapsto E \mid \Sigma * \Sigma \mid \dots \\ H &:= \Pi \wedge \Sigma \end{aligned}$$

Intuitively, in a symbolic heap $\Pi \wedge \Sigma$, the first conjunct Π contains only expressions describing the relations among program, primed and footprint variables given by the stack whereas Σ describes the allocated heap. The predicate $E \mapsto F$ is true when the cell E is allocated, its value is F , and nothing else is allocated. $\Sigma_1 * \Sigma_2$ holds when the heap can be split into components, one of which makes Σ_1 true and the other of which makes Σ_2 true. See [22]. We assume that primed variables in each symbolic heap H are existentially quantified.

⁷ Our requirement of a complete lattice and monotonicity can be weakened if we include a widening operator.

The use of \dots is to allow for various other predicates, such as for list segments and for trees. In this sense, the present section is setting down a parameterized analysis which can be instantiated, e.g., by [3, 5, 16]. More importantly, we are emphasizing that our footprint analysis algorithm (in the next section) is not tied to any of these particular analyses.

We define a “separation logic-based shape analysis” to consist of the following.

1. An instance $(S, \{S[E]\}_E, \text{rearr}, \text{filter}, \text{exec}, \text{abs})$ of the generic analysis from Section 3.
2. The shape analysis should use separation logic, in the style of [9]. This means that the underlying set S of the abstract domain consists of sets of symbolic heaps, and that for each expression E , all the symbolic heaps in the subset $S[E]$ of S are of the form $\Pi \wedge (E \mapsto F) * \Sigma$. We say that cell E is exposed by the `pointsto` relation.
3. A sound theorem prover \vdash for proving entailments between symbolic heaps.
4. For each symbolic heap $\Pi \wedge \Sigma$ in S and fresh footprint variable \bar{x} , the new symbolic heap $\Pi \wedge (E \mapsto \bar{x}) * \Sigma$ is in $S[E]$, or it can be shown to be inconsistent by the given theorem prover.
5. None of `rearr`, `filter`, `exec` and `abs` introduces new footprint variables into given symbolic heaps.
6. Writing G for the set of symbolic heaps in S containing only footprint variables, `abs` maps elements of G to G . Moreover, for all Π_0 with footprint variables only, if $\Pi \wedge \Sigma$ is in G , then $\Pi_0 \wedge \Pi \wedge \Sigma$ is in G , unless it is proved to be inconsistent by the theorem prover.

5 Footprint Analysis

Now suppose we are given a separation logic-based shape analysis as specified in the last section. Recall that S is the set of symbolic heaps and G is the set of symbolic heaps whose only free variables are footprint variables. Our footprint analysis is an instance of the generic analysis in Section 3, where the abstract domain of our algorithm is the topped powerset

$$\mathcal{P}^\top(S \times G).$$

A pair (H, F) in $S \times G$ represents the current heap H and the computed footprint F at the current program point. Note that the footprint can contain footprint variables only. The algorithm relies on this requirement to ensure that the computed footprint is a property of the initial states, rather than the states at the current program point.

We specify our algorithm by defining the data required by the generic analysis, which we call `newRearr`, `newFilter`, `newExec`, `newAbs` in order to avoid confusion with the abstract transfer functions of the given underlying shape analysis, which the footprint analysis will be defined in terms of.

First, we give the definition of `newRearr`, in terms of the rearrangement `rearr` of the given shape analysis:

$$\begin{aligned} \text{newRearr}(E) &: S \times G \rightarrow \mathcal{P}^\top(S[E] \times G) \\ \text{newRearr}(E)(H, F) &\stackrel{\text{def}}{=} \text{let } \mathcal{H} = \text{rearr}(E)(H) \\ &\quad \text{in if } \neg(\mathcal{H} = \top) \text{ then } \{(H', F) \mid H' \in \mathcal{H}\} \\ &\quad \text{else if } \neg(H \vdash E = \bar{x}_0 \text{ for some footprint var } \bar{x}_0) \text{ then } \top \\ &\quad \text{else if } (F * \bar{x}_0 \mapsto \bar{x}_1 \vdash \text{false for some fresh } \bar{x}_1) \\ &\quad \text{then } \top \\ &\quad \text{else } \{(H * E \mapsto \bar{x}_1, F * \bar{x}_0 \mapsto \bar{x}_1)\} \end{aligned}$$

This subroutine takes two symbolic heaps, H for the overapproximation of the reachable states and F for the footprint, and exposes a specified cell E from H . Intuitively, it first calls the rearrangement step of the underlying shape analysis to prove that a dereferenced cell E is allocated. In case this first attempt fails, the subroutine adds the missing cell to the footprint and the current symbolic heap. This is the point at which the underlying shape analysis would have stopped, reporting a fault. Note that before adding the points-to relation to F , the subroutine checks whether E can be rewritten in terms of a footprint variable \bar{x}_0 . This ensures that the computed footprint is independent of the values of variables whose value changes (program variables) or is determined during execution (primed variables).

Next, we define the subroutine `newFilter`:

$$\begin{aligned} \text{newFilter}(b) &: S \times G \rightarrow S \times G \\ \text{newFilter}(b)(H, F) &\stackrel{\text{def}}{=} \text{if } (\text{filter}(b)(H) \text{ is not defined}) \text{ then undefined} \\ &\quad \text{else let } H' = \text{filter}(b)(H) \\ &\quad \quad \text{in if } \neg(H \vdash b \Leftrightarrow \bar{b} \text{ for some } \bar{b} \text{ with footprint vars only}) \\ &\quad \quad \text{then } (H', F) \\ &\quad \quad \text{else } (H', \bar{b} \wedge F) \end{aligned}$$

This subroutine tries to rewrite b in terms of footprint variables only. If it succeeds, the rewriting gives an additional precondition \bar{b} that will make the test b hold: the computation can then pass through the filter, and the result of the rewriting is conjoined to the footprint. On the other hand, if the rewriting fails, the analyzer keeps the given footprint F .

Finally, the subroutine `newExec` is defined by the execution of `exec` for the first component H for shape invariants.

$$\text{newExec}(a[E])(H, F) \stackrel{\text{def}}{=} (\text{exec}(a[E])(H), F) \quad \text{newExec}(a)(H, F) \stackrel{\text{def}}{=} (\text{exec}(a)(H), F)$$

And `newAbs` is defined by applying abstraction to both the shape and footprint:

$$\text{newAbs}(H, F) \stackrel{\text{def}}{=} (\text{abs}(H), \text{abs}(F))$$

5.1 Hoare Triple Generation

We show how the footprint analysis algorithm can be used to generate true Hoare triples. First there is a pre-processing step which generates an initial symbolic

heap that saves the initial values of program variables into footprint variables. Then, after running footprint analysis, we run a post-processing step which takes the output of our algorithm and, for each computed precondition, it runs the underlying shape analysis to compute the appropriate postcondition.

Let x_1, \dots, x_n be program variables that appear in a given program c . Write $\llbracket - \rrbracket_f$ for our algorithm, and $\llbracket - \rrbracket_s$ for the given shape analysis. Formally, the Hoare triple generation for a program c works as follows:

let $\Pi_0 \stackrel{\text{def}}{=} (x_1 = \bar{x}_1 \wedge \dots \wedge x_n = \bar{x}_n)$ **and** $\mathcal{F} \stackrel{\text{def}}{=} \llbracket c \rrbracket_f(\{\Pi_0 \wedge \text{emp}, \text{emp}\})$
in if $(\mathcal{F} = \top)$ **then** report the possibility of a catastrophic fault
else $\left\{ \{F'\}c \left\{ \bigvee_{H' \in \mathcal{H}} H' \right\} \mid (H, F) \in \mathcal{F} \wedge F' = \text{ren}(\Pi_0 \wedge F) \wedge \mathcal{H} = \llbracket c \rrbracket_s(\{F'\}) \wedge \mathcal{H} \neq \top \right\}$.

Here $\text{ren}(\Pi_0 \wedge F)$ renames all the footprint variables by primed variables.

If the underlying shape analysis is sound with respect to a concrete semantics of a programming language then we automatically get true Hoare triples. However, it would be easy to generate *some* true Hoare triples, if we were content to generate precondition **false**. What our algorithm is aiming at is to generate preconditions that cover as many “safe states” as possible, ones which ensure that the program will not commit a memory fault. There can be, of course, no perfect such algorithm for computability reasons. In our case, though, it is well to mention two possible sources of inaccuracy.

First, because the analysis applies abstraction to the footprint (the eventual precondition), this can lead us outside of the safe states (it is essentially *weakening* a precondition). We have found that it very often leads to safe preconditions in our experimental results. An intuitive reason for this is that the safety of typical list programs is often insensitive to the abstraction present in shape analyses. But, because this “often” is not “always”, as we will see in the next section, the Hoare triple generation just described filters out these unsafe pre-states by calling the (assumed to be sound) underlying shape analysis.

Second, our algorithm does not perform as much case analysis on the structure of heap as is theoretically possible, and this leads to incompleteness (where fewer safe states are described than might otherwise be). We have made this choice for efficiency reasons. We believe that our experimental results in the next section show that this is not an unrealistic engineering decision. But we also discuss an example ([append. c](#)) where the resulting incompleteness arises.

Finally, we point out that from true Hoare triples computed by our analysis, one can easily construct a relational overapproximation of the transition relation of a program. Suppose that our analysis generated a set $\{\{P_i\}c\{Q_i\}\}_{i \in I}$ of true Hoare triples for a given program c . Then, by [6], there is a state transformer r (i.e., relation from **States** to **States** \cup **{wrong}**) with the following three properties: (1) the transformer r satisfies triple $\{P_i\}r\{Q_i\}$ for all $i \in I$; (2) it satisfies the locality conditions in separation logic⁸; and (3) the transformer overapproximates all the other state transformers satisfying (1) and (2) (i.e., it is bigger than those state transformers according to the subset ordering.) Indeed, [6] gives

⁸ The locality conditions are safety monotonicity and frame property in [28].

an explicit definition of the transformer r .⁹ This transformer overapproximates the relational meaning of program c , since all the triples $\{P_i\}c\{Q_i\}$ hold for c and the meaning of c satisfies the locality conditions.

6 Experimental Results

Our experimental results are for an implementation of our analysis developed using the CIL infrastructure [19]. We used two abstract domains for the experiments, one based on the simple list domain in [9] and the other with the domain of [2] which uses a higher-order variant of the list segment predicate to describe composite structures.

List program examples. Table 1 shows the results of applying the footprint analysis to a set of list programs taken from the literature.¹⁰ The Disjuncts column reports the number of disjuncts of the computed preconditions. Amongst all the computed preconditions, some can be unsafe and there can be redundancy in that one can imply another. The Unsafe Pre column indicates the preconditions filtered out when we re-execute the analysis. In the Discovered Precondition column we have dropped the redundant cases and used implication to obtain a compact representation that could be displayed in the table. For the same precondition, the table shows different disjuncts on different lines. For all tests except one (`merge.c`, discussed below) our analysis produced a precondition from which the program can run safely, without generating a memory fault, obtaining a true Hoare triple. We comment on a few representative examples.

`delete-doublestar` uses the usual C trick of double indirection to avoid unnecessary checking for the first element, when deleting an element from a list.

```
void delete-doublestar(nodeT **listP, elementT value)
{
  nodeT *currP, *prevP;
  prevP=NULL;
  for (currP=*listP; currP!=NULL; prevP=currP, currP=currP->next) {
    if (currP->element==value) { /* Found it. */
      if (prevP==NULL) *listP=currP->next;
      else prevP->next=currP->next;
      free(currP);
    } } }
```

⁹ Formally, $r \subseteq \text{States} \times (\text{States} \cup \{\text{wrong}\})$ is defined by:

$$\begin{aligned} (s, h)[r]\text{wrong} &\iff \forall i \in I. (s, h) \notin \llbracket P_i * \text{true} \rrbracket \\ (s, h)[r](s', h') &\iff \forall i \in I. \forall h_0, h_1. (s, h_0) \in \llbracket P_i \rrbracket \wedge h_0 \bullet h_1 = h \\ &\implies \exists h'_0. (s', h'_0) \in \llbracket Q_i \rrbracket \wedge h'_0 \bullet h_1 = h'. \end{aligned}$$

where $\llbracket P_i \rrbracket, \llbracket Q_i \rrbracket$ are the usual meaning of assertions and \bullet is a partial heap-combining operator in separation logic.

¹⁰ In some cases the reported memory consumption was exactly the same for different programs; this happens because the memory chunks allocated by OCAML's runtime system are too coarse to observe small differences between example programs.

The first disjunct of the discovered precondition is

```
listP|->x_ * ls(x_,x1_) * x1_->x2_
```

This shows the cells that are accessed when the element being searched for happens to be in the list. Note that it does not record list items which might follow the value: they are not accessed.¹¹ A postcondition for this precondition has just a list segment running to `x2_`:

```
listP|->x_ * ls(x_,x2_)
```

The other precondition

```
listP|->x_ * ls(x_,NULL)
```

corresponds to when the element being searched for is not in the list.

The algorithm fails to discover a circular list in the precondition

```
listP|->x_ * ls(x_,x_)
```

The program infinite loops on this input, but does not commit a memory safety violation. This is an example of incompleteness in our algorithm.¹²

Further issues can be seen by contrasting `append.c` and `append-dispose.c`. The former is the typical algorithm for appending two lists `x` and `y`. The computed precondition is

```
ls(x_,NULL)
```

Again, notice that nothing reachable from `y` is included, as the appended list is not traversed by the algorithm: it just swings a pointer from the end of the first list. However, when we post-compose appending with code to delete all elements in the acyclic list rooted at `x`, which is what `append-dispose.c` does, then the footprint requires an acyclic list from `y` as well

```
x|->x_ * ls(x_,NULL) * y|->y_ * ls(y_,NULL)
```

The only program for which we failed to find a safe precondition was `merge.c`, the usual program to merge two sorted lists: instead, footprint analysis returned all unsafe disjuncts (which were pruned at re-execution time). The reason is that our analysis essentially assumes that the safety of the program is insensitive to the abstraction performed in the analysis, and this is false for `merge.c`.

¹¹ This point could be relevant to interprocedural analysis, where [23, 10] pass a useful but coarse overapproximation of the footprint to a procedure, consisting of all abstract nodes reachable from certain roots.

¹² Note that the problem here does not have to do with circular lists *per se*, as our algorithm succeeds in finding preconditions for algorithms for circular linked lists (e.g., `traverse-circ.c`); rather, it has to do with incompleteness arising from avoidance of case analysis mentioned in Section 5.1.

Program	Time (s)	Memory	# of Disjuncts	Unsafe Pre	Discovered Precondition
append.c	0.03501	0.74Mb	4	0	$ls(x, NULL)$
append-dispose.c	0.09966	0.74Mb	17	0	$ls(x, NULL) * ls(y, NULL)$
copy.c	0.03076	0.74Mb	4	0	$ls(c, NULL)$
create.c	0.01370	0.49Mb	1	0	emp
delete-doublestar.c	0.04521	0.49Mb	10	0	$listP \mapsto x * ls(x, x1) * x1 \mapsto element: value,$ $listP \mapsto x * ls(x, NULL)$
delete-all.c	0.01357	0.49Mb	4	0	$ls(c, NULL)$
delete-all-circular.c	0.01564	0.49Mb	3	0	$c \mapsto c * ls(c, c)$
delete-lseg.c	0.63947	1.23Mb	48	0	$z \neq NULL \wedge ls(c, z) * ls(z, NULL),$ $z \neq w \wedge ls(c, z) * ls(z, w) * w \mapsto NULL,$ $z \neq w \wedge w \neq NULL \wedge ls(c, z) * ls(z, w) * w \mapsto w,$ $z \neq c \wedge c \mapsto NULL,$ $z \neq c \wedge z \neq c \wedge c \mapsto c * ls(c, NULL),$ $c = NULL \wedge emp$
find.c	0.05659	0.74Mb	12	0	$ls(c, b) * b \mapsto NULL,$ $b \neq NULL \wedge ls(c, b) * b \mapsto b,$ $b \neq c \wedge b \neq c \wedge c \mapsto c * lseg(c, NULL),$ $b \neq c \wedge c \mapsto NULL,$ $c = NULL \wedge emp$
insert.c	0.17049	0.74Mb	10	0	$e1 \neq NULL \wedge e2 \neq NULL \wedge c \neq d \wedge c \mapsto c * ls(c, d) * d \mapsto dta: e3,$ $e1 \neq NULL \wedge e2 \neq NULL \wedge c \neq NULL \wedge c \mapsto c * ls(c, NULL),$ $e1 \neq NULL \wedge c \mapsto NULL,$ $e1 \neq NULL \wedge e2 = NULL \wedge c \mapsto c * c \mapsto -,$ $e1 = NULL \wedge c \mapsto -,$ $c = NULL \wedge emp$
merge.c	0.56092	1.47Mb	30	30	—
reverse.c	0.01965	0.74Mb	4	0	$ls(c, NULL)$
traverse-circ.c	0.01322	0.49Mb	3	0	$c \mapsto c * ls(c, c)$

Table 1. Experimental results for list programs.

IEEE 1394 firewire driver routines. We then changed the abstract domain in our implementation, swapping the simple list domain for the domain from [2]. Table 2 reports experimental results on several routines from a firewire driver for Windows.¹³ We emphasize that the ability of that domain to analyze the driver code is not a contribution of the present paper: it was already shown in [2] when preconditions were generated by environment code or supplied manually. Here, we are just using that domain with our footprint analysis algorithm.

The procedure `t1394Diag_PnpRemoveDevice`, for which our analysis timed out, has five while loops, two of which are nested, and multiple nested conditionals. At the time of writing, our analysis does not implement several optimizations

¹³ After dropping redundant disjunct and simplify by implication, the precondition for device drivers remain still considerably large. Therefore, for space limitation, in this table, we do not report the discovered preconditions.

Program	Time (s)	Memory	# of Disjuncts	Unsafe Pre
t1394Diag-CancelIrp.c	0.08928	1.23Mb	11	2
t1394Diag-CancelIrpFix.c	0.20461	1.23Mb	10	0
t1394Diag_PnpRemoveDevice	T/O	—	—	—
t1394-BusResetRoutine.c	0.14924	1.23Mb	4	0
t1394-GetAddressData.c	0.08692	1.23Mb	9	2
t1394-GetAddressDataFix.c	0.08906	1.23Mb	3	0
t1394-IsochDetachCompletionRoutine.c	1.76640	2.70Mb	39	0
t1394-SetAddressData.c	0.06614	1.23Mb	9	1
t1394-SetAddressDataFix.c	0.12242	1.23Mb	9	0

Table 2. Experimental results from firewire device driver routines.

for scalability. For example, we have not yet implemented the acceleration techniques based on widening from [5].

For five (out of nine) of these routines our analysis found only sound preconditions from which it is ensured the program will run safely. For three of these routines (`t1394Diag_CancelIrp`, `t1394_GetAddressData`, `t1394_SetAddressData`) for which it was known to have memory errors (see [2] for details), our analysis found two kinds of preconditions:

- *Safe preconditions* that exclude the errors. The analyzer generated true Hoare triples for these preconditions.
- *Unsafe preconditions* that lead to (in this case) known memory errors. For these analyzed routines the memory errors occur when they are given empty lists. All of these unsafe empty-list cases are included in the discovered preconditions. But, they are the only reasons for the preconditions to be unsafe; if we semantically rule out these empty-list cases from these preconditions by altering them manually, the preconditions cover only safe states as can be confirmed by re-execution.

The errors were fixed in `t1394Diag_CancelIrpFix`, `t1394_GetAddressDataFix` and `t1394_SetAddressDataFix`, such that the routines run safely even for the empty-list cases. The analysis correctly discovered this fact, by computing safe preconditions that include empty-list cases (in addition to all the other cases in the safe preconditions for the original routines).

7 Conclusion

We have presented a shape analysis that is able to discover preconditions, and we have presented initial experimental results. We are not aware of another published shape analysis that discovers preconditions (which is why we have not compared our analysis or experimental results to other work in shape analysis).

We have used two abstract domains in our experiments, one for simple linked lists and another for composite structures [9, 2], but others could be used as

well as long as they possess the basic separation logic structure that drives our analysis. Several other abstract domains based on separation logic formulae have been described [3, 5, 16, 12], and we expected that other shape domains that have appeared (e.g., [25, 15, 13, 4, 18, 1]) could be modified to have the requisite structure. This would not require using separation logic formulae literally, but rather needs the abstract domain to reflect the the partial commutative monoid of heap composition used in its semantics (as was done in [26, 17]).

One of the basic ideas used in our analysis, that of abstracting preconditions as well as postconditions, could conceivably be replayed for other abstract domains than our shape domains. It would require more work to investigate for any given domain. We emphasize, though, that this general idea is not a significant contribution of the present paper, and we make no claims about application to other sorts of domain (e.g., numerical domains). Rather, the main contribution is the way that the footprint idea is used to design a particular family of shape analyses that discover preconditions, and the demonstration that some of the resulting analyses can possess a non-trivial degree of precision.

Footprint analysis has potential benefits for speeding up and improving the accuracy of interprocedural and concurrency shape analyses, and for encapsulation. With the footprint analysis one might analyze several procedures independently, and then use the results as (partial) summaries to avoid (certain) recomputations in an (even whole program) interprocedural analysis [23, 10]. A thread-modular concurrency analysis has recently been defined [11]. The logic upon which it is based [20] requires preconditions for concurrent processes, but in [11] this issue is skirted by assigning the empty heap as a precondition to each concurrent process: the ideas here might be used to extend that analysis. We add that there are numerous technical problems to be overcome for this potential to be realized, such as the right treatment of cutpoints [23] together with footprints.

A further off possible application is when the calling context (“main” program) is not even available, or very large (e.g., an operating system): one might try to use footprint analysis to analyze code fragments that would otherwise require a whole-program analysis. Our experiments give initial indications on such an idea, but more work is needed to evaluate its ultimate viability. Conversely, there is the persistent problem of analyzing programs that themselves call other unknown or as yet unwritten procedures. It would be conceivable to use footprint analysis to treat the unknown procedures as “black holes”, where one starts footprint analysis again after a black hole to discover a precondition for the code that comes after; this would then function as a postcondition for the procedure call itself.

We do not mean to imply that the use of footprint analyses in these areas is in any way straightforward, and only hope that this work might help to spur further developments towards obtaining truly modular shape analyses.

Acknowledgments. We are grateful to Byron Cook, Noam Rinetzky and Mooly Sagiv for discussions on the ideas in this paper. This research was supported by EPSRC.

References

1. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *6th VMCAI*, pages 164–180, 2005.
2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O’Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. To appear in *CAV’07*.
3. J. Berdine, B. Cook, D. Distefano, and P. O’Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *18th CAV*, pages 386–400, 2006.
4. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract tree regular model checking of complex dynamic data structures. In *13th SAS*, pages 52–70, 2006.
5. C. Calcagno, D. Distefano, P.W. O’Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *13th SAS*, pages 182–203, 2006.
6. C. Calcagno, P. O’Hearn, and H. Yang. Local action and abstract separation logic. To appear in *LICS’07*, 2007.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pp238-252, 1977.
8. P. Cousot and R. Cousot. Modular static program analysis. In *11th CC*, pages 159–178, 2002.
9. D. Distefano, P. O’Hearn, and H. Yang. A local shape analysis based on separation logic. In *12th TACAS*, pages 287–302, 2006.
10. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *13th SAS*, pages 240–260, 2006.
11. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. To appear in *PLDI 2007*.
12. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. To appear in *PLDI 2007*.
13. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *32nd POPL*, pages 310–323, 2005.
14. B. Jeannot, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. Tech. Rep. 1505, Comp. Sci. Dept., Univ. of Wisconsin, 2004.
15. Tal Lev-Ami, Neil Immerman, and Mooly Sagiv. Abstraction for shape analysis with fast and precise transformers. In *18th CAV*, pages 547–561. 2006.
16. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in Separation Logic for imperative list-processing programs. In 3rd SPACE Workshop, 2006.
17. R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *13th TACAS*, 2007.
18. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *6th VMCAI*, pages 181–198, 2005.
19. G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL:intermediate language and tools for analysis and transformation of C programs. In *11th CC*, pages 213–228, 2002.
20. P. O’Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 2007. To appear, preliminary version appeared in CONCUR’04.
21. P. O’Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*, pp1–19, 2001.

22. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp 55-74, 2002.
23. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *32nd POPL*, pp296-309, 2005.
24. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1-50, 1998.
25. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3valued logic. *ACM TOPLAS*, 24(3):217-298, 2002.
26. Élodie-Jane Sims. An abstract domain for separation logic formulae. In *Proceedings of the 1st International Workshop on Emerging Applications of Abstract Interpretation (EAAI06)*, pages 133-148, 2006.
27. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *12th ESOP*, pages 204-222, 2003.
28. H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *5th FOSSACS*, LNCS 2303, 2002.