A Local Shape Analysis based on Separation Logic

Dino Distefano[1], Peter W. O'Hearn[1], and Hongseok Yang[2]

[1] Queen Mary, University of London
[2] Seoul National University

Abstract. We describe a program analysis for linked list programs where the abstract domain uses formulae from separation logic.

1 Introduction

A shape analysis attempts to discover the shapes of data structures in the heap at program points encountered during a program's execution. It is a form of pointer analysis which goes beyond the tabulation of shallow aliasing information (e.g., can these two variables be aliases?) to deeper properties of the heap (e.g., is this an acyclic linked list?).

The leading current shape analysis is that of Sagiv, Reps and Wilhelm, which uses very generic and powerful abstractions based on three-valued logic [17]. Although powerful, a problem with this shape analysis is that it behaves in a global way. For example, when one updates a single abstract heap cell this may require also the updating of properties associated with all other cells. Furthermore, each update of another cell might itself depend on the whole heap. This global nature stems from the use of certain instrumentation predicates, such as ones for reachability, to track properties of nodes in the heap: an update to a single cell might alter the value of a host of instrumentation predicates.

In contrast, separation logic provides an approach to reasoning about the heap that has a strong form of locality built in [14]. Typically, one reasons about a collection of cells in isolation, and their update does not necessitate checking or updating cells that are held in a different component of a separating conjunction. It thus seems reasonable to try to use ideas from separation logic in program analysis, with an eye towards the central problem of modularity in the analysis.

Our technical starting point is recent work of Berdine, Calcagno and O'Hearn [5], who defined a method of symbolic execution of certain separation logic formulae called symbolic heaps. Their method is not, by itself, suitable as an abstract semantics because there are infinitely many symbolic heaps and there is no immediate way to guarantee convergence of fixed-point calculations. Here, we obtain a suitable abstract domain by working with (a variation on) their method of symbolic execution, and adding to it an abstraction or widening operator which converts any symbolic heap to one in a certain "canonical form". This abstraction method is an adaptation of work in [7, 8] to the symbolic heaps of Berdine et. al. In contrast to unrestricted symbolic heaps we show that there are only finitely many canonical forms, resulting in termination of the fixed-point calculation used in the abstract semantics of while loops.

Our abstract domain uses linked lists only. Other abstractions based on separation logic might be considered as well.

After defining the analysis we turn to locality. We describe a sense in which the abstract semantics obeys the Frame Rule of separation logic, and we identify a notion of footprint as an input-output relation that mentions only those symbolic heap cells accessed by a program. The footprint provides a sound over-approximation of a program's entire (abstract) meaning. The results on locality give a way to automatically infer sound answers for large states from those obtained on small ones as input, suggesting further possible developments in interprocedural and concurrency analyses.

1.1 Related Work

In work on heap analysis (see [15] for discussion) much use has been made of a "storeless semantics" where the model is built from equivalence classes of paths rather than locations. The storeless semantics has the pleasant property that it is garbage collecting by its very nature, but it is also extremely complex. This makes it highly nontrivial to see that a particular analysis based on it is sound. In contrast, here we work directly with a store model, and soundness is almost immediate. The abstraction we use is defined by rewrite rules which are all true implications in separation logic, and the symbolic execution rules are derived from true Hoare triples.

Recent work on shape analysis [15, 16] might be regarded as taking some steps towards separation logic. Early on in separation logic there was an emphasis on what was referred to as "local reasoning": reasoning concentrates on the cells accessed during computation [12]. In [15, 16] an interprocedural analysis is described where a procedure summary is constructed which involves only the (abstract) cells reachable from input parameters or variables free in a procedure. The method of applying a procedure does not, however, explicitly utilize a separating conjunction operator $*$; one might say that the general idea of local reasoning is adopted (or altered), but the formal apparatus of separation logic is not.

In this paper we reciprocate by taking some steps towards shape analysis. Our intention initially was full reciprocation: to build an interprocedural analysis. But, after labouring for the better part of a year, we decided to aim lower: to define an abstract domain and abstract post operator, together with an account of its locality, for a language without procedures. In doing this we have been influenced by shape analysis, but have not adopted the formal apparatus of shape graphs or 3-valued logic. We hope that this paper can serve as a springboard for further developments in local interprocedural and modular concurrency analysis.

We want to make clear that we do not claim that our analysis is superior, in a practical sense, to existing shape analyses. Although it works well on small examples, we have not yet demonstrated that it scales to large programs. Also, from a methodological point of view, in the framework of [17] different abstractions are obtained in a uniform way, where a notion of "canonical abstraction" results once instrumentation predicates are nailed down. In contrast, here we have just one particular set of rewrite rules that have been hand-built; how this might be turned into a more general scheme is not obvious.

Nonetheless, we believe that research on how separation logic, or more particularly, the local reasoning idea, might be used in program analysis is of interest because it suggests a genuinely different approach which has promise for the central problem of obtaining modular analyses. A very good example of this is the recent work of Amtoft et. al. [2, 1] which uses local reasoning in information flow analysis (this is a more shallow form of analysis than shape analysis, but they are successful in formulating a very modular analysis).

Finally, in work carried out independently of (and virtually in parallel to) that here, Magill et. al. have defined a method of inferring invariants for linked list programs in separation logic [9]. They also utilize a symbolic execution mechanism related to [5], and give rewrite rules to attempt to find fixed points. There are many detailed differences: (i) they use a different basic list predicate than we do and, as they point out, have difficulty dealing with acyclic lists, where that is a strong point of our analysis; (ii) they do predicate abstraction of arithmetic operations, where we do not; (iii) and they use an embedding into Presburger arithmetic to help decide implications and Hoare triples, where we do not provide a method for deciding implications (or Hoare triples); (iv) their algorithm does not always terminate, where ours does. But, there is remarkable similarity.

2 Semantic Setting

We first describe the general semantic setting for this work. Following the framework of abstract interpretation [6], we will work with complete lattices D: The semantics of a command c will be given by a continuous function $[\![c]\!]\colon D \to D$.

If we are given a programming language with certain primitive operations p, together with conditionals, sequencing and while loops, then to define the semantics we must specify the meaning $[\![p]\!]$ of each primitive operation as well as a continuous function, $\mathsf{filter}(b)\colon D \to D$, for each boolean. Typically, D is built from subsets of a set of states, and the filter function removes those elements that are not consistent with b's truth. The semantics extends to the rest of the language in the usual way.

$$[\![c\,;c']\!] = [\![c]\!]\,;[\![c']\!] \qquad [\![\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c']\!] = (\mathsf{filter}(b)\,;[\![c]\!]) \sqcup (\mathsf{filter}(\neg b)\,;[\![c']\!])$$

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \lambda d.\ \mathsf{filter}(\neg b)\ \ \mathit{fix}\ \lambda d'.\ d \sqcup (\mathsf{filter}(b)\,;[\![c]\!])(d')$$

One way to understand the semantics of **while** is to view d' as a loop invariant. The d in the lhs of $\sqcup$ means that the loop invariant d' should be implied by the precondition, and the rhs of $\sqcup$ means that d' is preserved by the body. (Here, the fixed-point operator has been moved inward from its usual position in semantics, so that it applies to predicates instead of two command denotations.)

Our domains D will be constructed using a powerset operation. If S is a set we denote by $\mathcal{P}(S)$ the "topped" powerset of S, that is, the set of subsets of

$S \cup \{\top\}$. Here, $\top \notin S$ is a special element that corresponds to memory fault (accessing a dangling pointer). If we were to take logical implications between elements of $\mathcal{P}(S)$ into account then we would make $\{\top\}$ the top and equate all sets containing $\top$. For simplicity in this paper we just use the subset order.

Given a relation $(p \Longrightarrow) \subseteq S \times (S \cup \{\top\})$, with membership notated $\sigma, p \Longrightarrow \sigma'$, we can lift it to a function $p^\dagger \colon \mathcal{P}(S) \to \mathcal{P}(S)$ by

$$p^\dagger X = \{\sigma' \mid \exists \sigma \in X.\, (\sigma, p \Longrightarrow \sigma') \text{ or } (\sigma = \sigma' = \top)\}.$$

The semantics of primitive commands will be given by first specifying an execution semantics $\Longrightarrow$ and then lifting it to $\mathcal{P}(S)$.

Every semantics we work with will have two additional properties: that $\{\top\}$ is mapped to $\{\top\}$ and that it preserves unions. Because of this we could in fact work with a corresponding map $[\![c]\!]_\dagger \colon S \to \mathcal{P}(S)$ instead of $[\![c]\!] \colon \mathcal{P}(S) \to \mathcal{P}(S)$.

3 Concrete and Symbolic Heaps

Throughout this paper we assume a fixed *finite* set Vars of program variables (ranged over by $x, y, \ldots$), and an infinite set Vars' of primed variables (ranged over by $x', y', \ldots$). The primed variables will not be used within programs, only within logical formulae (where they will be implicitly existentially quantified).

Definition 1. *A* symbolic heap $\Pi \mid \Sigma$ *consists of a finite set* Π *of equalities and a finite set* Σ *of heap predicates. The equalities* $E{=}F$ *are between* expressions E *and* F*, which are variables* x*, or primed variables* x'*, or* nil*. The elements of* Σ *are of the form*

$$E \mapsto F \qquad \mathsf{ls}(E, F) \qquad \mathsf{junk}.$$

We use $\mathcal{SH}$ *to denote the set of* consistent *symbolic heaps. (For the definition of consistency, see below.)*

The first two heap predicates are "precise" in the sense of [13]; each cuts out a unique piece of (concrete) heap. The points-to assertion $E \mapsto F$ can hold only in a singleton heap, where E is the only active cell. Similarly, when a list segment holds of a given heap, the path it traces out is unique, and goes through all the cells in the heap. This precise nature of the predicates is helpful when accounting for deallocation. For each symbolic heap $\Pi \mid \Sigma$, we call Π *pure part* of $\Pi \mid \Sigma$, and Σ *spatial part* of $\Pi \mid \Sigma$.

The junk predicate is used in the canonicalization phase of our analysis to swallow up garbage. It is crucial for termination of our analysis, and it has the useful property to reveal memory leaks.

Besides the heap formulae, symbolic heaps also keep track of equalities involving pointer variables and nil.

We often use the notation $\Sigma * P$ for the (disjoint) union of a formula P onto the spatial part of a symbolic heap, and we similarly use $\Pi \wedge P$ in the pure part.

The meaning of a symbolic heap corresponds to a formula

$$\exists x_1' x_2' \ldots x_n'.\ \bigwedge\nolimits_{P \in \Pi} P \ \wedge\ \bigstar_{Q \in \Sigma} Q \ ,$$

in separation logic, where $\{x'_1, \ldots, x'_n\}$ is the set of all the primed variables in Σ and Π. More formally, the meaning of a symbolic heap is given by a satisfaction relation $s, h \vDash \Pi \mid \Sigma$, where s is a stack and h a (concrete) heap.

$$\begin{array}{ll} \mathsf{Values} = \mathsf{Locations} \cup \{\mathsf{nil}\} & \mathsf{Heaps} = \mathsf{Locations} \rightharpoonup_f \mathsf{Values} \\ \mathsf{Stacks} = (\mathsf{Vars} \cup \mathsf{Vars}') \to \mathsf{Values} & \mathsf{States} = \mathsf{Stacks} \times \mathsf{Heaps} \end{array}$$

The semantics is given in Table 1. The operation $h_0 * h_1$ there is the union of heaps with disjoint domains. We give the semantics for the singleton sets in the pure and spatial parts, and then for unions. There, the clause for list segments is given informally, but corresponds to the least predicate satisfying

$$\mathsf{ls}(E, F) \iff E{\neq}F \wedge (E{\mapsto}F \vee (\exists x'.E{\mapsto}x' \ * \ \mathsf{ls}(x', F))).$$

Table 1 Semantics of Symbolic Heaps

$$\mathcal{C}[\![x]\!]s = s(x) \qquad \mathcal{C}[\![x']\!]s = s(x') \qquad \mathcal{C}[\![\mathsf{nil}]\!]s = \mathsf{nil}$$

$s, h \vDash \{\}$	iff h is the empty heap []
$s, h \vDash \{E{\mapsto}F\}$	iff $h = [\mathcal{C}[\![E]\!]s{\mapsto}\mathcal{C}[\![F]\!]s]$
$s, h \vDash \{\mathsf{ls}(E, F)\}$	iff there is a nonempty acyclic path from $\mathcal{C}[\![E]\!]s$ to $\mathcal{C}[\![F]\!]s$ in h and this path contains all heap cells in h
$s, h \vDash \{\mathsf{junk}\}$	iff $h \neq \emptyset$
$s, h \vDash \Sigma_0 * \Sigma_1$	iff $\exists h_0, h_1.\ h = h_0 * h_1$ and $s, h_0 \vDash \Sigma_0$ and $s, h_1 \vDash \Sigma_1$
$s \vDash \{\}$	always
$s \vDash \{E{=}F\}$	iff $\mathcal{C}[\![E]\!]s = \mathcal{C}[\![F]\!]s$
$s \vDash \Pi_0 \cup \Pi_1$	iff $s \vDash \Pi_0$ and $s \vDash \Pi_1$
$s, h \vDash \Pi \mid \Sigma$	iff $\exists \boldsymbol{v'}.\ (s(\boldsymbol{x'} \mapsto \boldsymbol{v'}) \vDash \Pi)$ and $(s(\boldsymbol{x'} \mapsto \boldsymbol{v'}), h \vDash \Sigma)$ where $\boldsymbol{x'}$ is the collection of primed variables in $\Pi \mid \Sigma$

Our analysis will require us to be able to answer some questions about symbolic heaps algorithmically: whether two expressions are equal, whether they are unequal, whether the heap is inconsistent, and whether a cell is allocated.

$$\begin{array}{ll} \Pi \vdash E{=}F & \Pi \mid \Sigma \vdash E{\neq}F \ \ (\text{when } \mathsf{Vars}'(E, F) = \emptyset) \\ \Pi \mid \Sigma \vdash \mathsf{false} & \Pi \mid \Sigma \vdash \mathtt{Allocated}(E) \ \ (\text{when } \mathsf{Vars}'(E) = \emptyset) \end{array}$$

$\Pi \vdash E{=}F$ is easy to check. It just considers whether E and F are in the same equivalence class induced by the equalities in Π. The other operators use subroutine allocated, which takes Σ and an expression E, and decides whether Σ implies that E points to an allocated cell, by a "nontrivial reason": allocated ignores the case where Σ is not satisfiable and implies all formulae.

$$\mathsf{allocated}(\Sigma, E) = \exists E'.\ (E \mapsto E' \in \Sigma) \ \text{ or } (\mathsf{ls}(E, E') \in \Sigma).$$

We then define the other querying operators as follows:

$$\begin{array}{rcl} \Pi \mathrel{¦} \Sigma \vdash \mathsf{false} & \iff & (\exists E.\ \Pi \vdash E{=}\mathsf{nil} \text{ and } \mathsf{allocated}(\Sigma, E)), \text{ or} \\ & & (\exists E, F.\ \Pi \vdash E{=}F \text{ and } \mathsf{ls}(E, F) \in \Sigma), \text{ or} \\ & & \left.\begin{array}{l} \exists E, F.\ \Pi \vdash E{=}F \text{ and } \Sigma \text{ contains two distinct} \\ \quad \text{predicates whose lhs's are, respectively, } E \text{ and } F \end{array}\right) \end{array}$$

$$\begin{array}{rcl} \Pi \mathrel{¦} \Sigma \vdash E{\neq}F & \iff & (E{=}F \wedge \Pi \mathrel{¦} \Sigma) \vdash \mathsf{false} \\ \Pi \mathrel{¦} \Sigma \vdash \mathtt{Allocated}(E) & \iff & \Pi \mathrel{¦} \Sigma \vdash \mathsf{false}, \text{ or } (\exists E'.\ \Pi \vdash E{=}E' \text{ and } \mathsf{allocated}(\Sigma, E')) \end{array}$$

These definitions agree with what one would obtain from a definition in terms of the satisfaction relation $\models$, but they are simple syntactic checks that do not require calling a theorem prover.

The rules that define our analysis will preserve consistency of symbolic heaps (that $\Pi \mathrel{¦} \Sigma \nvdash \mathsf{false}$). In particular, inconsistent heaps introduced in branches of if statements or as a result of tests in a while loop will be filtered out.

4 Concrete and Symbolic Execution Semantics

The grammar of commands for the programming language used in this paper is given by

$$\begin{array}{ll} b ::= E{=}E \mid E{\neq}E & \\ p ::= x{:=}\,E \mid x{:=}\,[E] \mid [E]{:=}\,F \mid \mathbf{new}(x) \mid \mathbf{dispose}(E) & \text{Primitive Commands} \\ c ::= p \mid c\,;c \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c & \text{Commands} \end{array}$$

We do not consider commands that contain any primed variables amongst their expressions. We include only a single heap dereferencing operator $[\cdot]$ which refers to the "next" field. In the usual way, our experimental implementation ignores commands that access fields other than "next" (say, a data field), and treats any boolean conditions other than those given as nondeterministic.

4.1 Concrete Semantics

The execution rules for the primitive commands are as follows, where in the faulting rule (the last rule) we use notation for primitive commands that access heap cell E:

$$A(E) ::= [E]{:=}\,F \mid x{:=}\,[E] \mid \mathbf{dispose}(E)$$

Concrete Execution Rules

$$\frac{\mathcal{C}[\![E]\!]s = n}{s, h,\ x{:=}\,E \implies (s \mid x \mapsto n), h} \qquad \frac{\mathcal{C}[\![E]\!]s = \ell \quad h(\ell) = n}{s, h,\ x{:=}\,[E] \implies (s \mid x \mapsto n), h}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell \quad \mathcal{C}[\![F]\!]s = n \quad \ell \in \mathsf{dom}(h)}{s, h,\ [E]{:=}\,F \implies s, (h \mid \ell \mapsto n)} \qquad \frac{\ell \notin \mathsf{dom}(h)}{s, h,\ \mathbf{new}(x) \implies (s \mid x \mapsto \ell), (h \mid \ell \mapsto n)}$$

$$\frac{\mathcal{C}[\![E]\!]s = \ell}{s, h * [\ell \mapsto n],\ \mathbf{dispose}(E) \implies s, h} \qquad \frac{\mathcal{C}[\![E]\!]s \notin \mathsf{dom}(h)}{s, h,\ A(E) \implies \top}$$

Notice the tremendous amount of nondeterminism in **new**: it picks out *any* location not in the domain of the heap, and *any* value n for its contents.

The concrete semantics is given in the topped powerset $\mathcal{P}(\mathsf{States})$, where the filter map is

$$\mathsf{filter}(b)X \;=\; \{(s,h) \in X \mid \mathcal{C}[\![b]\!]s \;=\; \text{true}\} \cup \{\top \mid \top \in X\}$$

where $\mathcal{C}[\![b]\!]s \in \{\text{true,false}\}$ just checks equalities by looking up in the stack s.

With these definitions we may then set $\mathcal{C}[\![p]\!] \;=\; p^\dagger$ and by the recipe of Section 2 we obtain the concrete semantics, $\mathcal{C}[\![c]\!] \colon \mathcal{P}(\mathsf{States}) \to \mathcal{P}(\mathsf{States})$, of every command c.

4.2 Symbolic Semantics

The symbolic execution semantics $\sigma, A \Longrightarrow \sigma'$ takes a symbolic heap σ and a primitive command, and transforms it into an output symbolic heap or $\top$. In these rules we require that the primed variables x', y' be fresh.

Symbolic Execution Rules

$$\begin{array}{lll}
\Pi \mid \Sigma, & x{:=}\,E & \Longrightarrow x{=}E[x'/x] \wedge (\Pi \mid \Sigma)[x'/x] \\
\Pi \mid \Sigma * E{\mapsto}F, & x{:=}\,[E] & \Longrightarrow x{=}F[x'/x] \wedge (\Pi \mid \Sigma * E{\mapsto}F)[x'/x] \\
\Pi \mid \Sigma * E{\mapsto}F, & [E]{:=}\,G & \Longrightarrow \Pi \mid \Sigma * E{\mapsto}G \\
\Pi \mid \Sigma, & \mathbf{new}(x) & \Longrightarrow (\Pi \mid \Sigma)[x'/x] * x{\mapsto}y' \\
\Pi \mid \Sigma * E{\mapsto}F, & \mathbf{dispose}(E) & \Longrightarrow \Pi \mid \Sigma
\end{array}$$

$$\frac{\Pi \mid \Sigma \not\vdash \mathtt{Allocated}(E)}{\Pi \mid \Sigma,\ A(E) \Longrightarrow \top}$$

Rearrangement Rules

$$P(E,F) \;::=\; E{\mapsto}F \mid \mathsf{ls}(E,F)$$

$$\frac{\Pi_0 \mid \Sigma_0 * P(E,G),\ A(E) \Longrightarrow \Pi_1 \mid \Sigma_1}{\Pi_0 \mid \Sigma_0 * P(F,G),\ A(E) \Longrightarrow \Pi_1 \mid \Sigma_1}\ \Pi_0 \vdash E{=}F$$

$$\frac{\Pi_0 \mid \Sigma_0 * E{\mapsto}x' * \mathsf{ls}(x',G),\ A(E) \Longrightarrow \Pi_1 \mid \Sigma_1}{\Pi_0 \mid \Sigma_0 * \mathsf{ls}(E,G),\ A(E) \Longrightarrow \Pi_1 \mid \Sigma_1} \qquad \frac{\Pi \mid \Sigma * E{\mapsto}F,\ A(E) \Longrightarrow \Pi' \mid \Sigma'}{\Pi \mid \Sigma * \mathsf{ls}(E,F),\ A(E) \Longrightarrow \Pi' \mid \Sigma'}$$

The execution rules that access heap cell E are stated in a way that requires their pre-states to explicitly have $E{\mapsto}F$. Sometimes the knowledge that E is allocated is less explicit, such as in $\{E{=}x\} \mid \{x{\mapsto}y\}$ or $\mathsf{ls}(E,F)$, and we use rearrangement rules to put the pre-state in the proper form. The first rearrangement rule simply makes use of equalities to recognize that a dereferencing step is possible, and the other two correspond to unrolling a list segment.

In contrast to the concrete semantics, the treatment of allocation is completely deterministic (up to renaming of primed variables). However, a different kind of nondeterminism results in rearrangement rules that unroll list segments.

All that is left to define the symbolic (intermediate) semantics $\mathcal{I}[\![c]\!] \colon \mathcal{P}(\mathcal{SH}) \to \mathcal{P}(\mathcal{SH})$ by the recipe before is to define the filter map. It adds the equality for the $E{=}F$ case, but does not do so for the $E{\neq}F$ case because we do not have inequalities in our symbolic domain.

$$\begin{aligned}\mathsf{filter}(E{=}F)X &= \{\top \mid \top{\in}X\} \cup \{(E{=}F \wedge \Pi \mid \Sigma) \mid \Pi \mid \Sigma \in X \text{ and } \Pi \mid \Sigma \not\vdash E{\neq}F\}\\ \mathsf{filter}(E{\neq}F)X &= \{\top \mid \top{\in}X\} \cup \{(\Pi \mid \Sigma) \in X \mid \Pi \not\vdash E{=}F \text{ and } \Pi \mid \Sigma \not\vdash \mathsf{false}\}\end{aligned}$$

To state the sense in which the symbolic semantics is sound we define the "meaning function" $\gamma\colon \mathcal{P}(\mathcal{SH}) \to \mathcal{P}(\mathsf{States})$:

$$\gamma(X) = \text{if } (\top \in X) \text{ then } (\mathsf{States} \cup \{\top\}) \text{ else } (\{(s,h) \mid \exists \Pi \mid \Sigma \in X.\ (s,h) \models \Pi \mid \Sigma\})$$

Theorem 2. *The symbolic semantics is a sound overapproximation of the concrete semantics:* $\forall X \in \mathcal{P}(\mathcal{SH}).\ \mathcal{C}[\![c]\!](\gamma(X)) \subseteq \gamma(\mathcal{I}[\![c]\!]X)$.

5 The Analysis

The domain $\mathcal{SH}$ of symbolic heaps is infinite. Even though there are finitely many program variables, primed variables can be introduced during symbolic execution. For example, in a loop that includes allocation we can generate formulae $x{\mapsto}x' * x'{\mapsto}x'' \cdots$ of arbitrary length.

In order to ensure fixed-point convergence we perform abstraction. The abstraction we consider is specified by a collection of rewrite rules which perform abstraction by gobbling up primed variables. This is done by merging lists, swallowing single cells into lists, and abstracting two cells by a list. We also remove primed variables from the pure parts of formulae, and we collect all garbage into the predicate junk.

5.1 Canonicalization Rules

The canonicalization rules are reported in Table 2. We again use the notation $P(E,F)$ to stand for an atomic formula either of the form $E{\mapsto}F$ or $\mathsf{ls}(E,F)$.

The most important rules are the last two. The sense of abstraction that these rules implement is that we ignore any facts that depend on a midpoint in a list segment, unless it is named by a program variable. There is a subtlety in interpreting this statement, however. One might perhaps have expected the last rule to leave out the $P_3(G,H)$ $*$-conjunct, but this would result in unsoundness; as Berdine and Calcagno pointed out [4, 5] (our abstraction rules are obtained from their proof rules), we must know that the end of a second list segment does not point back into the first if we are to concatenate them. We are forced, by considerations of soundness, to keep some primed midpoints, such as in the formula $\mathsf{ls}(x,x') * \mathsf{ls}(x',y)$, to which no rewrite rule applies.

Notice the use of a $\cup$ rather than a $*$ on the rhs of the (Gb1) and (Gb2) rules. This has the effect that when more than one unreachable node named by a primed variable is present, all of them get put into the unique junk node.

Table 2 Abstraction Rules

$$\frac{}{E{=}x' \wedge \Pi \mid \Sigma \rightsquigarrow (\Pi \mid \Sigma)[E/x']}\ (\mathsf{St1}) \qquad \frac{}{x'{=}E \wedge \Pi \mid \Sigma \rightsquigarrow (\Pi \mid \Sigma)[E/x']}\ (\mathsf{St2})$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma)}{\Pi \mid \Sigma * P(x', E) \rightsquigarrow \Pi \mid \Sigma \cup \mathsf{junk}}\ (\mathsf{Gb1}) \quad \frac{x', y' \notin \mathsf{Vars}'(\Pi, \Sigma)}{\Pi \mid \Sigma * P_1(x', y') * P_2(y', x') \rightsquigarrow \Pi \mid \Sigma \cup \mathsf{junk}}\ (\mathsf{Gb2})$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma, E, F) \qquad \Pi \vdash F{=}\mathsf{nil}}{\Pi \mid \Sigma * P_1(E, x') * P_2(x', F) \rightsquigarrow \Pi \mid \Sigma * \mathsf{ls}(E, \mathsf{nil})}\ (\mathsf{Abs1})$$

$$\frac{x' \notin \mathsf{Vars}'(\Pi, \Sigma, E, F, G, H) \quad \Pi \vdash F{=}G}{\Pi \mid \Sigma * P_1(E, x') * P_2(x', F) * P_3(G, H) \rightsquigarrow \Pi \mid \Sigma * \mathsf{ls}(E, F) * P_3(G, H)}\ (\mathsf{Abs2})$$

5.2 The Algorithm

We say that $\Pi \mid \Sigma$ is a canonical symbolic heap if it is consistent (i.e., $\Pi \mid \Sigma \nvdash \mathsf{false}$) and no canonicalization rule applies to it, and we denote by $\mathcal{CSH}$ the set of all such. We can immediately observe:

Lemma 3 (Strong Normalization). $\rightsquigarrow$ *has no infinite reduction sequences.*

This, together with the results in the next section, would be enough to define a terminating analysis. But, there are many distinct reduction sequences and to try all of them in an analysis would lead to a massive increase in nondeterminism. We have not proven a result to the effect that choosing different reduction sequences matters in the final result (after applying the meaning function γ), but neither have we found examples where the difference can be detected. So, in our implementation we have chosen a specific strategy which applies the equality rules, followed by ($\mathsf{Gb1}$), followed by abstraction rules, followed by ($\mathsf{Gb2}$). In the theory, we just presume that we have a function (rather than relation)

$$\mathsf{can} \colon \mathcal{SH} \to \mathcal{CSH}$$

which takes a symbolic heap $\Pi \mid \Sigma$ and returns a canonical symbolic heap $\Pi' \mid \Sigma'$ where $\Pi \mid \Sigma \rightsquigarrow^* \Pi' \mid \Sigma'$.

[We remark that $\mathsf{can}(\Pi \mid \Sigma)$ is not the best (logically strongest) canonical heap implied by $\Pi \mid \Sigma$. A counterexample is $\{\} \mid \{x{\mapsto}x', x'{\mapsto}y, y{\mapsto}\mathsf{nil}\}$. This symbolic heap is reduced to $\{\} \mid \{\mathsf{ls}(x, y), y{\mapsto}\mathsf{nil}\}$ by the canonicalization, but implies another symbolic heap $\{\} \mid \{x{\mapsto}x', x'{\mapsto}z', y{\mapsto}\mathsf{nil}\}$, which is not (logically) weaker than $\{\} \mid \{\mathsf{ls}(x, y), y{\mapsto}\mathsf{nil}\}$. We believe that this "problem" is fixable; we conjecture that there is a preorder $\sqsubseteq$ on $\mathcal{SH}$ such that (i) $\sqsubseteq$ is a sub preorder of the logical implication and (ii) $\mathsf{can}(\Pi \mid \Sigma)$ is the smallest canonical heap greater than or equal to $\Pi \mid \Sigma$ with respect to $\sqsubseteq$. As of this writing we have not succeeded in proving this conjecture. If true, it would perhaps open the way to a study pinpointing where precision is and is not lost (as in, e.g., [3]) using Galois connections. Although valuable, such questions are secondary to our more basic aim of existence (soundness and termination) of the analysis.]

Let $\mathtt{in} \colon \mathcal{P}(\mathcal{CSH}) \to \mathcal{P}(\mathcal{SH})$ denote the inclusion function. We define the abstract semantics for each primitive command p by the equation

$$\mathcal{A}[\![p]\!] \;=\; \mathtt{in}\,; \mathcal{I}[\![p]\!]\,; (\mathsf{can}^\dagger).$$

The filtering map in the abstract semantics is just the restriction of the symbolic one to $\mathcal{CSH}$. Then, by the recipe from Section 2 we obtain a semantics

$$\mathcal{A}[\![c]\!] : \mathcal{P}(\mathcal{CSH}) \to \mathcal{P}(\mathcal{CSH})$$

for every command.

The soundness of the abstract semantics relies on the soundness of the rewriting rules.

Lemma 4 (Soundness of $\rightsquigarrow$). *If $\Sigma \mathrel{\dot{|}} \Pi \rightsquigarrow \Sigma' \mathrel{\dot{|}} \Pi'$ then $\Sigma \mathrel{\dot{|}} \Pi \vdash \Sigma' \mathrel{\dot{|}} \Pi'$.*

The statement of soundness of the abstract semantics is then the same as for the symbolic semantics, except that we quantify over $\mathcal{P}(\mathcal{CSH})$ instead of $\mathcal{P}(\mathcal{SH})$.

Theorem 5. *The abstract semantics is a sound overapproximation of the concrete semantics: $\forall X \in \mathcal{P}(\mathcal{CSH}).\ \mathcal{C}[\![c]\!](\gamma(X)) \subseteq \gamma(\mathcal{A}[\![c]\!]X)$.*

Here are some examples of running the analysis on particular pre-states, taken from an implementation of it in OCaml.

Example 1. This is the usual program to reverse a list. Here 0 is used to denote nil, $x{\to}tl$ is used instead of $[x]$, and the commas in the analysis results are replaced by the corresponding logical connectives.

`Program:` $p{:=}0\,;$ **while** $(c{\neq}0)$ **do** $(n{:=}c{\to}tl\,; c{\to}tl{:=}p\,; p{:=}c\,; c{:=}n)$

`Pre:` $\{\}\mathrel{\dot{|}}\{\mathsf{ls}(c,0)\}$ `Post:` $\{c{=}0 \wedge c{=}n \wedge n{=}0\}\mathrel{\dot{|}}\{\mathsf{ls}(p,0)\} \ \vee\ \{c{=}0 \wedge c{=}n \wedge n{=}0\}\mathrel{\dot{|}}\{p{\mapsto}0\}$

`Inv:` $\{p{=}0\}\mathrel{\dot{|}}\{\mathsf{ls}(c,0)\} \ \vee\ \{c{=}n \wedge n{=}0\}\mathrel{\dot{|}}\{p{\mapsto}0\} \ \vee\ \{c{=}n \wedge n{=}0\}\mathrel{\dot{|}}\{\mathsf{ls}(p,0)\} \ \vee$
$\{c{=}n\}\mathrel{\dot{|}}\{p{\mapsto}0 * \mathsf{ls}(n,0)\} \ \vee\ \{c{=}n\}\mathrel{\dot{|}}\{\mathsf{ls}(p,0) * \mathsf{ls}(n,0)\}$

Given a linked list as a precondition, the analysis calculates that the postcondition might be a linked list or a single points-to fact. The postcondition has some redundancy, in that we could remove the second disjunct without affecting the meaning; this is because we have used the subset ordering on sets of states, rather than one based on implication. The analysis also calculates the pictured loop invariant, which captures that p and c point to separated linked lists.

Running the analysis to the same program with a circular linked list as input gives the following (we omit the calculated invariant, which has 11 disjuncts).

`Pre:` $\{\}\mathrel{\dot{|}}\{\mathsf{ls}(c,c') * \mathsf{ls}(c',c)\}$

`Post:` $\{c{=}0 \wedge c{=}n \wedge n{=}0\}\mathrel{\dot{|}}\{p{\mapsto}p' * \mathsf{ls}(p',p)\} \ \vee\ \{c{=}0 \wedge c{=}n \wedge n{=}0\}\mathrel{\dot{|}}\{p{\mapsto}p' * p'{\mapsto}p\}$

Example 2. This is the program to dispose a list.

`Program:` **while** $(c{\neq}0)$ **do** $(t{:=}c\,; c{:=}\,c{\to}tl\,;$ **dispose**$(t))$

`Pre:` $\{\}\mathrel{\dot{|}}\{\mathsf{ls}(c,0)\}$ `Post:` $\{c{=}0\}\mathrel{\dot{|}}\{\}$ `Inv:` $\{c{=}0\}\mathrel{\dot{|}}\{\} \ \vee\ \{\}\mathrel{\dot{|}}\{\mathsf{ls}(c,0)\}$

The spatial part $\{\}$ of the postcondition expresses that the heap is empty on termination. If we leave out the dispose instruction, it returns postcondition $\{c{=}\mathsf{nil}\}\mathrel{\dot{|}}\{t{\mapsto}\mathsf{nil} * \mathsf{junk}\}$ (showing memory leak). When we run the analysis on this program on a circular list or $\mathsf{ls}(c,d)$ it reports a memory fault.

In addition to these examples we have run the analysis on a range of other small programs, such as list append, list copy, programs to insert and delete from the middle of a list, programs to delete and filter from circular lists, and to delete a segment between two values from a sorted list. The execution times ranged from a few milliseconds for reverse, copy and append to three seconds for delete-a-segment (running on a 1.5GHz PowerBook G4), and in space requirements none of them exceeded the OCaml default initial heap size of 400kB.

In coverage of examples, and in the nature of the abstraction itself, the analysis here appears to be somewhat similar to the one reported in [10]. A careful study of this relationship could be worthwhile.

6 Termination

Although the abstract semantics exists, we have not yet established that the algorithm it determines always terminates. We do that by showing that the domain $\mathcal{CSH}$, consisting of the normal forms of the rewriting rules, is finite.

To gain some insight into the nature of the canonical symbolic heaps here are some examples, where the pure part Π is empty (and left out).

Irreducible	**Reducible**
$\mathsf{ls}(x,x') * \mathsf{ls}(y,x') * \mathsf{ls}(x',\mathsf{nil})$	$\mathsf{ls}(x,x') * \mathsf{ls}(x',y') * \mathsf{ls}(y',\mathsf{nil})$
$\mathsf{ls}(x,x') * \mathsf{ls}(x',x)$	$\mathsf{ls}(x,y') * \mathsf{ls}(y',x') * \mathsf{ls}(x',x)$
$\mathsf{ls}(x,x')$	$\mathsf{ls}(x',x)$
$\mathsf{ls}(x,x') * \mathsf{ls}(x',y)$	$\mathsf{ls}(x,x') * \mathsf{ls}(x',y) * \mathsf{ls}(y,z)$

In the first element of the first row, variable x' is shared (pointed to by x and y), and this blocks the application of rule ($\mathsf{Abs1}$) because of its variable condition. On the other hand, the second element can be reduced, in fact twice, to end up with $\mathsf{ls}(x,\mathsf{nil})$. The second row contains two cycles, one of (syntactic) length two and the other of length three. The first of these cannot be reduced. We would need to know that $x{=}\mathsf{nil}$ to apply ($\mathsf{Abs1}$) and we cannot, because x in $\mathsf{ls}(x,x')$ cannot be nil or else we would have an inconsistent formula. The second in this row can, however, be reduced, to the first. In the third row x' is a reachable variable that possibly denotes a dangling pointer and there is no way to eliminate it. In the second it is not reachable, and can be removed using the ($\mathsf{Gb1}$) rule. Note that this removal is sound, because all heap predicates, including $\mathsf{ls}(x',x)$, imply junk. In the final row, first x' points to a possibly dangling variable y. We cannot remove x', because transforming $\mathsf{ls}(x,x') * \mathsf{ls}(x',y)$ to $\mathsf{ls}(x,y)$ is unsound; when $y = x = 10$, no heap can satisfy $\mathsf{ls}(x,y)$, while a cycle from location 10 of length 2 satisfies $\mathsf{ls}(x,x') * \mathsf{ls}(x',y)$. The rule ($\mathsf{Abs2}$) is arranged to prevent this unsoundness. If we tack on another heap formula to ensure that y does not point to any internal cells of the list segment $\mathsf{ls}(x,x')$, then ($\mathsf{Abs2}$) can apply.

Based on these ideas we can characterize the normal forms of $\leadsto^*$ using "graphical" ideas of path and reachability, as well as conditions about sharing, cycles, and dangling pointers.

Definition 6. *1. A* path *in $\Pi \mid \Sigma$ is a sequence of expressions $E_0, E_1, \ldots, E_n$ such that*

$$\forall i \in \{1, \ldots, n\}.\ \exists E, E'.\ \Pi \vdash E_{i-1}{=}E \text{ and } \Pi \vdash E_i{=}E' \text{ and } P(E, E') \in \Sigma.$$

Reachability between expressions is defined in the usual way: E is reachable from *E'* in *$\Pi \mid \Sigma$ if and only if there is a path in $\Pi \mid \Sigma$ that starts from E and ends in E'.*

2. An expression E in $\Pi \mid \Sigma$ is shared *if and only if Σ contains two distinct elements $P_0(E_0, E_0')$ and $P_1(E_1, E_1')$ such that $\Pi \vdash E{=}E_0'$ and $\Pi \vdash E{=}E_1'$.*

3. A primed variable x' in a cycle (a path from E to itself) is an internal node *if and only if it is not shared.*

4. E is called possibly dangling *in $\Pi \mid \Sigma$ if and only if*

*(a) $\Pi \nvdash E{=}$*nil*,*

(b) there exists some E' such that $\Pi \vdash E{=}E'$ and E' is the second argument of some heap predicate in Σ, and

(c) there are no expressions F' such that $\Pi \vdash E{=}F'$ and F' is the first argument of some heap predicate in Σ.

5. E points to a possibly dangling expression *if and only if there are E', F such that $\Pi \vdash E{=}E'$, $P(E', F) \in \Sigma$, and F possibly dangles.*

Definition 7 (Reduced Symbolic Heap). *A symbolic heap $\Pi \mid \Sigma$ is* reduced *if and only if*

1. *Π does not contain primed variables;*
2. *every primed variable x' in Σ is reachable from some unprimed variable; and*
3. *for every reachable variable x', either (a) x' is shared, or (b) x' is the internal node of a cycle of length precisely two, or (c) x' points to a possibly dangling variable, or (d) x' is possibly dangling.*

In (b) of this definition the length refers to the syntactic length of a path, not the length of a denoted cycle. For example, $\mathsf{ls}(x, x') * \mathsf{ls}(x', x)$ has syntactic length two, even though it denotes cycles of length two or greater.

This definition of reduced heaps is not particularly pretty; its main point is to give us a way to prove termination of our analysis.

Proposition 8 (Canonical Characterization). *When a symbolic heap $\Pi \mid \Sigma$ is consistent, $\Pi \mid \Sigma$ is reduced if and only if $\Pi \mid \Sigma \not\rightsquigarrow$.*

We consider the formulae in $\mathcal{CSH}$ as being equivalent up to renaming of primed variables. With this convention, we can show $\mathcal{CSH}$ finite.

Proposition 9. *$\mathcal{CSH}$ is finite.*

The proof of this proposition proceeds by first showing a lemma that bounds the number of primed variables in any reduced form. In essence, the condition 3 of the definition of "reduced" stops there being infinitely many possible primed variables (starting from a fixed finite set of program variables). This then limits the number of atomic formulas that can appear, giving us finiteness. The overall bound one obtains is exponential (for the record, we have an argument that gives a very coarse bound of $2^{(129n^2+18n+2)}$). This the leads us to

Theorem 10. *The algorithm specified by $\mathcal{A}[\![\cdot]\!]$ always terminates.*

7 Locality

We now describe locality properties of the semantics, beginning with an example. Suppose that we have a queue, represented in memory as a list segment from c to d. An operation for getting an element is

$$x{:=}c\,;\,c{:=}c{\to}tl \qquad \text{/* get from left of queue, put in } x \text{ */}$$

The list segment might not be the whole storage, of course. In particular, we might have an additional element pointed to by d which is (perhaps) used to place an element into the queue. When we run our tool on an input reflecting this state of affairs we obtain

$$\texttt{Pre:}\ \{\} \mid \{\mathsf{ls}(c,d) * d{\mapsto}d'\} \qquad \texttt{Post:}\ \{c{=}d\} \mid \{x{\mapsto}d * d{\mapsto}d'\} \vee \{\} \mid \{x{\mapsto}c * \mathsf{ls}(c,d) * d{\mapsto}d'\}$$

However, it is clear that the $d \mapsto d'$ information is irrelevant, that a run of the tool on the smaller input gives us all the information we need.

$$\texttt{Pre:}\ \{\} \mid \{\mathsf{ls}(c,d)\} \qquad \texttt{Post:}\ \{c{=}d\} \mid \{x{\mapsto}d\} \ \vee\ \{\} \mid \{x{\mapsto}c * \mathsf{ls}(c,d)\}$$

In fact, the behaviour of the tool in the first case follows from that in the second, using the Frame Rule of separation logic. This example is motivated by the treatment of a concurrent queue in [11]. The fact that we do not have to consider the cell d when inserting is crucial for a verification which shows that the two ends of a nonempty queue can be manipulated concurrently. To produce such results from an analysis, rather than a by-hand proof, we would similarly like to avoid the need to analyze the entire state including the cell d.

We can give a theoretical account of the locality of our analysis using the following notions. First, we define a notion of $*$ on entire symbolic heaps.

$$(\Pi_1 \mid \Sigma_1) * (\Pi_2 \mid \Sigma_2) \ = \ (\Pi_1 \cup \Pi_2 \mid \Sigma_1 * \Sigma_2).$$

This is a partial operation, which is undefined when $\Sigma_1 * \Sigma_2$ is undefined, or when $(\Pi_1 \cup \Pi_2 \mid \Sigma_1 * \Sigma_2)$ is inconsistent, or when some primed variable appears both in $\Pi_1 \mid \Sigma_1$ and in $\Pi_2 \mid \Sigma_2$. We extend this to $\mathcal{SH} \cup \{\top\}$ by stipulating $(\Pi \mid \Sigma) * \top \ = \ \top \ = \ \top * (\Pi \mid \Sigma)$. It then lifts to a total binary operation on $\mathcal{P}(\mathcal{SH})$ by

$$X * Y = \{\sigma_1 * \sigma_2 \mid \sigma_1 \in X,\, \sigma_2 \in Y\}.$$

To formulate the locality property we suppose a fixed set Mod of modified variables, that appear to the left of := or in $\mathbf{new}(x)$ in a given command c.

Theorem 11 (Frame Rule). *For all $X, Y \in \mathcal{P}(\mathcal{CSH})$, if $\mathsf{Vars}(Y) \cap Mod = \emptyset$ then $\gamma(\mathcal{A}[\![c]\!](X * Y)) \subseteq \gamma((\mathcal{A}[\![c]\!]X) * Y)$.*

There are two reasons why we get an overapproximation $\subseteq$ rather than exact match here. First, and trivially, there might be states in X where c faults, returns $\top$, while it never does for states in $X{*}Y$. The second reason is best understood by example. When the program $\mathbf{new}(x)\,;(\mathbf{if}\ x{=}y\ \mathbf{then}\ z{:=}\,a\ \mathbf{else}\ z{:=}\,b)\,;\mathbf{dispose}(x)$ is run in the empty heap, it returns two post-states, one where $z{=}a$ and the another where $z{=}b$. But when run in $y{\mapsto}y'$ the **if** branch is ruled out and we only get $z{=}b \mid y{\mapsto}y'$ as a conclusion. However, we get $z{=}a \mid y{\mapsto}y'$ as an additional possibility starting from $y \mapsto y'$, when we put the small output together with $y \mapsto y'$ using $*$. Although precision can be lost when passing to smaller states, in many examples we have considered it is an acceptable loss or none.

For a given command c and symbolic heap σ we define

1. $\mathsf{safe}(c, \sigma)$ iff $\top \notin \mathcal{A}[\![c]\!]\{\sigma\}$
2. $\sigma_1 \preceq \sigma_3$ iff $\exists \sigma_2.\, \sigma_3 \;=\; \sigma_1 * \sigma_2$
3. $\sigma \prec \sigma'$ iff $\sigma \preceq \sigma'$ and $\sigma \neq \sigma'$
4. $\mathsf{onlyaccesses}(c, \sigma)$ iff $\mathsf{safe}(c, \sigma)$ and $\neg \exists \sigma' \prec \sigma.\, \mathsf{safe}(c, \sigma')$.

The notion of accesses is coarse. For example, $\mathsf{onlyaccesses}([x]{:=}\, y, \mathsf{ls}(x, \mathsf{nil}))$ holds, even though a single cell can be picked out of the list segment. A stronger notion of accesses, and hence footprint, might be formulated taking implications between symbolic heaps into account as well as $\preceq$.

The footprint is partial function $\mathsf{foot}(c) \colon \mathcal{CSH} \rightharpoonup \mathcal{P}(\mathcal{CSH})$,

$$\mathsf{foot}(c)\sigma \;=\; \text{if } (\mathsf{onlyaccesses}(c, \sigma)) \text{ then } (\mathcal{A}[\![c]\!]\{\sigma\}) \text{ else (undefined).}$$

The point of the footprint is that, as a set of pairs, it can be compact compared to the entire meaning. For the disposelist program in Example 2, the footprint has three entries, with preconditions $\{\} \mid \{\mathsf{ls}(c, \mathsf{nil})\}$, $\{\} \mid \{c \mapsto \mathsf{nil}\}$ and $\{c{=}\mathsf{nil}\} \mid \{\}$. The entire meaning has 16 entries, corresponding to the number of canonical symbolic heaps over a single input variable c.

To express the sense in which the footprint is a sound representation of the semantics of c we show how any potential footprint can be "fleshed out" by applying the idea behind the Frame Rule. Again, let Mod be the set of modified variables in a given command c, and for each $\Pi \mid \Sigma$, let $\mathsf{unaffectedEqs}(\Pi \mid \Sigma)$ be the set of equalities $E{=}F$ in Π such that $\mathsf{Vars}(E{=}F) \cap Mod = \emptyset$. If $f \colon \mathcal{CSH} \rightharpoonup \mathcal{P}(\mathcal{CSH})$, then $\mathsf{flesh}(f) \colon \mathcal{CSH} \to \mathcal{P}(\mathcal{CSH})$ is defined as follows:

$$\begin{array}{rcl}
\mathsf{validSplit}(\sigma_0, \sigma_1, \sigma) & \iff & \sigma_0 * \sigma_1 = \sigma \text{ and } \mathsf{Vars}(\sigma_1) \cap Mod = \emptyset \text{ and} \\
& & \sigma_0 \in \mathsf{dom}(f) \text{ and } \mathsf{unaffectedEqs}(\sigma_1) = \mathsf{unaffectedEqs}(\sigma) \\
\mathsf{flesh}(f)\sigma & = & \text{if } (\neg \exists \sigma_0, \sigma_1.\, \mathsf{validSplit}(\sigma_0, \sigma_1, \sigma)) \text{ then } \{\top\} \\
& & \text{else let } \sigma_0', \sigma_1' \text{ be symbolic heaps s.t. } \mathsf{validSplit}(\sigma_0', \sigma_1', \sigma) \\
& & \qquad \text{in } \mathcal{P}(\mathsf{can})(f(\sigma_0') * \{\sigma_1'\})
\end{array}$$

The fleshing out picks one access point, and adds as many $*$-separated invariants as possible to the access point.

Theorem 12. *The footprint is a sound overapproximation of the abstract semantics:* $\forall X \in \mathcal{P}(\mathcal{CSH}).\ \gamma(\mathcal{A}[\![c]\!]X) \subseteq \gamma(\mathsf{foot}(c)^{\dagger} X)$.

The calculation of whole footprints is, of course, not realistic. A more practical way to employ the footprint idea would be, given an input state σ, to look at substates on which a procedure or command does not produce a fault. In interprocedural analysis, we might record the input-output behaviour on as small states as possible when tabulating a procedure summary. This would be similar to [16], but would not involve entire reachable substates. In concurrency, we would look for disjoint substates of an input state on which to run parallel commands: if these input states were safe for the commands in question, then we could soundly avoid (many) interleavings during symbolic execution. We hope to report on these matters at a later time.

ACKNOWLEDGEMENTS. Thanks to Josh Berdine, Cristiano Calcagno, Ivana Mijajlovic, Andreas Podelski, Noam Rinetzky, Mooly Sagiv, Anindya Banergee, and the anonymous referees for helpful comments on this work. Yang was supported by R08-2003-000-10370-0 from the Basic Research Program of Korea Science & Engineering Foundation. Distefano and O'Hearn were supported by the EPSRC.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow analysis of pointer programs. 33rd POPL, to appear, 2006.

[2] T. Amtoft and A. Banerjee. Information flow analysis in logical form. 11th Static Analysis Symposium, LNCS3184, pp100-115, 2004.

[3] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *7th TACAS, LNCS*, 2031:268–283, 2001.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. Proceedings of FSTTCS, LNCS 3328, Chennai, December, 2004.

[5] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS 2005*, volume 3780 of *LNCS*, 2005.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. 4th ACM Symposium on Principles of Programming Languages. pages 238–252, 1977.

[7] D. Distefano. *On model checking the dynamics of object-based software: a foundational approach.* PhD thesis, University of Twente, 2003.

[8] D. Distefano, A. Rensink, and J.-P. Katoen. Who is pointing when to whom: on model-checking pointer structures. CTIT Technical Report TR-CTIT-03-12, Faculty of Informatics, University of Twente, March 2003.

[9] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in Separation Logic for imperative list-processing programs. Draft, July 2005, 2005.

[10] R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. *Proceedings of 6th VMCAI*, pp181-198, 2005.

[11] P. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 2006. to appear. Preliminary version appeared in CONCUR'04, LNCS 3170, 49–67.

[12] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. of 15th CSL*, LNCS, pages 1–19. Springer-Verlag, 2001.

[13] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st POPL*, pages 268–280, 2004.

[14] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp 55-74, 2002.

[15] N. Rinetzky, J. Bauer, T. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. *32nd POPL*, pp296–309, 2005.

[16] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th International Static Analysis Symposium (SAS)*, 2005.

[17] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.