Automatic Parallelization with Separation Logic

Cristiano Calcagno, Philippa Gardner, and Mohammad Raza*

Imperial College London

Abstract. We present a separation logic framework which can express properties of memory separation between different points in a program. We describe an algorithm based on this framework for determining independences between statements in a program which can be used for parallelization.

1 Introduction

Automatic parallelization techniques are generally based on a detection of independence between statements in a program, in the sense that two statements accessing separate resources can be executed in parallel. Such techniques have been extensively studied and successfully applied for programs with simple data types and arrays, but there has been limited progress for programs that manipulate pointers and dynamic data structures [9,10,7,11,13]. Separation logic is a recent approach to the study of pointer programs [15] in which the separation of resource is expressed with the logical connective '$*$'. This approach has been implemented in many program analysis tools for the purposes of shape analysis and safety verification [16,4,8,1]. However, these analyses cannot be used for program parallelization, because the $*$ connective only expresses separation of memory at a single program point and therefore cannot determine independences between statements in a program. In this paper we present an extended separation logic framework which can express memory separation properties throughout a program's lifetime.

The framework is based on an extension of separation logic formulae with *labels*, which are used to keep track of memory regions through an execution. Symbolic execution based on separation logic [2,5] is extended so that occurrences of the same label, even in different formulae referring to different program points, refer to the same memory locations throughout the execution. However, the symbolic execution mechanism is such that memory locations cannot always be represented by the same label through an entire execution: fresh labels have to be introduced during the execution to replace existing labels and the new labels may represent memory regions that overlap with old ones. For this reason, we keep an *intersection log* which relates labels that may represent possibly overlapping memory regions. To keep track of the memory locations that are accessed by a command, we keep a *footprint log* which records the labels of the part of the call-site formula that the command depends on. These labels are clearly determined for primitive commands. For procedure calls and while loops, the labels are determined by a frame inference method [2] that keeps track of the labels by using a form of *label respecting* entailment between formulae.

* principle author

The main issue in demonstrating the soundness of our approach is in the semantic interpretation of labels. This must be given in the context of an entire execution rather than on a formula that describes a single state. For this we adapt the action trace semantics of programs introduced in [6] to prove soundness of concurrent separation logic. We introduce the notion of a symbolic execution trace, and a notion of satisfaction between an action trace and a symbolic execution trace formalises the relation between labels in a symbolic execution and locations in a concrete execution.

Our approach fits in the line of work of using static analysis to collect independence facts between program points. Our departure point is the use of separation logic-based shape analysis. One of the strengths of our approach is that it does not rely on *reachability* properties of data structures to detect independences, as in [10,7]. The approach here is more precise in that it finds the *footprints* of commands: the cells that are actually accessed rather than the cells that may possibly be accessed.

In this paper we illustrate our method in a restricted setting adapted from [2], working with simple list and tree formulae. Our proposed method is engineered so that it can be applied as a post-processing phase starting from the output of an existing shape analysis based on separation logic, and requires only minor changes to existing symbolic execution engines. We begin in the next section by introducing labelled symbolic heaps, which are standard symbolic heap formulae extended with labels. In the next section we describe the programming language we work with and an intermediate language which is actually used in the analysis. We then describe the extended symbolic execution algorithm for determining independences, and discuss examples. In the following section we describe the frame inference method that keeps track of the labels in the inferred frame axiom. In the final section we demonstrate the soundness of the method with respect to an action trace semantics of programs.

2 Labelled Symbolic Heaps

The concrete heap model is based on a set of fields $\mathtt{Fields}$, and disjoint sets $\mathtt{Loc}$ of locations and $\mathtt{Val}$ of non-addressable values, with $\mathtt{nil} \in \mathtt{Val}$. We assume a finite set $\mathtt{Var}$ of program variables and an infinite set $\mathtt{Var}'$ of primed variables. Primed variables will not be used in programs, only within the symbolic heaps where they will be implicitly existentially quantified. We then set $\mathtt{Heaps} = \mathtt{Loc} \rightharpoonup_f (\mathtt{Fields} \to \mathtt{Val} \cup \mathtt{Loc})$ and $\mathtt{Stacks} = (\mathtt{Var} \cup \mathtt{Var}') \to \mathtt{Val} \cup \mathtt{Loc}$. We work with a class of separation logic formulae called *symbolic heaps*, as described in [2,5], except that we introduce *labels*, $l \in \mathtt{Lab}$, on the simple spatial formulae in symbolic heaps.

$x, y, .. \in \mathtt{Var}$	program variables
$x', y', .. \in \mathtt{Var}'$	primed variables
$l, k.. \in \mathtt{Lab}$	labels
$f_1, f_2, .. \in \mathtt{Fields}$	fields

$$
\begin{array}{rll}
E, F ::= & \mathtt{nil} \mid x \mid x' & \text{expressions} \\
\rho ::= & f_1 : E_1, ..., f_k : E_k & \text{record expressions} \\
P ::= & E = E \mid \neg P & \text{simple pure formulae} \\
\Pi ::= & \mathtt{true} \mid P \mid \Pi \wedge \Pi & \text{pure formulae} \\
S ::= & E \mapsto [\rho] \mid \mathtt{ls}(E, F) \mid \mathtt{dlls}(E_f, E_b, F_f, F_b) \mid \mathtt{tree}(E) & \text{simple spatial formulae} \\
\Sigma ::= & \mathtt{emp} \mid \langle S \rangle_l \mid \Sigma * \Sigma & \text{spatial formulae with labelled simple conjuncts} \\
\mathtt{SH} ::= & \Pi \,\vert\, \Sigma & \text{symbolic heaps}
\end{array}
$$

We include an *empty* label $\bullet \in \mathtt{Lab}$ for situations where the label is unspecified. Except for the empty label $\bullet$, we require that every label has at most a unique occurence in a symbolic heap. We let $\mathtt{Lab}(\Pi \,\vert\, \Sigma)$ denote the set of labels in the symbolic heap $\Pi \,\vert\, \Sigma$. Labels shall be used to keep track of separate portions of resource through the execution of a program. They are therefore not given an interpretation on a symbolic heap, but shall be interpreted in the context of a symbolic execution. The interpretation of symbolic heaps is otherwise standard, given by a forcing relation $s, h \models A$ where $s \in \mathtt{Stacks}$, $h \in \mathtt{Heaps}$, and A is a pure assertion, spatial assertion, or symbolic heap. $h = h_0 * h_1$ indicates that the domains of h_0 and h_1 are disjoint, and h is their graph union. We assume the fields $n, b, l, r \in \mathtt{Fields}$, where n is the next field for list segments, b is the back field for doubly linked segments, and l and r are the left and right fields for trees.

$$[\![x]\!]s = s(x) \quad [\![x']\!]s = s(x') \quad [\![\mathtt{nil}]\!]s = \mathtt{nil}$$

$$
\begin{array}{lll}
s, h \models E_1 = E_2 & \text{iff} & [\![E_1]\!]s = [\![E_2]\!]s \\
s, h \models \neg P & \text{iff} & s, h \not\models P \\
s, h \models \mathtt{true} & & \text{always} \\
s, h \models \Pi_0 \wedge \Pi_1 & \text{iff} & s, h \models \Pi_0 \text{ and } s, h \models \Pi_1 \\
s, h \models \langle E_0 \mapsto [f_1 : E_1, \ldots, f_k : E_k] \rangle_l & \text{iff} & h = [[\![E_0]\!]s \rightharpoonup r] \text{ where } r(f_i) = [\![E_i]\!]s \text{ for } i \in 1..k \\
s, h \models \langle \mathtt{ls}(E, F) \rangle_l & \text{iff} & \text{there is a linked list segment from } E \text{ to } F \\
s, h \models \langle \mathtt{dlls}(E_f, E_b, F_f, F_b) \rangle_l & \text{iff} & \text{there is a doubly linked list segment from } E_f \text{ to } F_f \\
 & & \text{with initial and final back pointers } E_b \text{ and } F_b \\
s, h \models \langle \mathtt{tree}(E) \rangle_l & \text{iff} & \text{there is a tree at } E \\
s, h \models \mathtt{emp} & \text{iff} & h = \emptyset \\
s, h \models \Sigma_0 * \Sigma_1 & \text{iff} & \exists h_0 h_1.\, h = h_0 * h_1 \text{ and } s, h_0 \models \Sigma_0 \text{ and } s, h_1 \models \Sigma_1 \\
s, h \models \Pi \,\vert\, \Sigma & \text{iff} & \exists v. s(x' \mapsto v), h \models \Pi \text{ and } s(x' \mapsto v), h \models \Sigma \\
 & & \text{where } x' \text{ is the collection of primed variables in } \Pi | \Sigma
\end{array}
$$

The formal semantics of the data structure formulae is given as the least predicates satisfying the following inductive definitions:

$$
\begin{array}{rl}
\mathtt{ls}(E, F) \iff & (E = F \wedge \mathtt{emp}) \vee (E \neq F \wedge \exists y. E \mapsto [n : y] * \mathtt{ls}(y, F)) \\
\mathtt{dlls}(E_f, E_b, F_f, F_b) \iff & (E_f = F_f \wedge E_b = F_b \wedge \mathtt{emp}) \vee \\
 & (E_f \neq F_f \wedge E_b \neq F_b \wedge \exists y. E_f \mapsto [n : y, b : E_b] * \mathtt{dlls}(y, E_f, F_f, F_b)) \\
\mathtt{tree}(E) \iff & (E = \mathtt{nil} \wedge \mathtt{emp}) \vee (\exists x, y. E \mapsto [l : x, r : y] * \mathtt{tree}(x) * \mathtt{tree}(y))
\end{array}
$$

3 Programming Language

We consider a standard programming language with procedures.

$b ::= E = E \mid E \neq E$	boolean expressions
$A ::= x := E \mid x := E \to f \mid E_1 \to f := E_2 \mid \texttt{new}(x)$	atomic commands
$c ::= i : A \mid i : f(\overrightarrow{E_1}; \overrightarrow{E_2}) \mid i : \texttt{if}\ b\ c_1\ c_2 \mid i : \texttt{while}\ b\ c \mid c_1; c_2$	indexed commands ($i \in I$)
$p ::= . \mid f(\vec{x}; \vec{y})\{\texttt{local}\ \vec{z}; c\}; p$	programs

A program is given by a number of procedure definitions. We assume that every command $i : c$ in a procedure body has a unique index i from some set of indices I. We let $I(c)$ be the set of indices of all command statements in c. In a procedure with header $f(\vec{x}; \vec{y})$, $\vec{x} = x_1, .., x_n$ are the variables not changed in the body, and $\vec{y} = y_1, .., y_m$ are the variables that are assigned to. We assume that all variables occuring free in the body are declared in the header. We define $\texttt{free}(c)$ and $\texttt{mod}(c)$ sets as the set of free and modified variables of c. For atomic commands these are defined as usual. For procedures we have $\texttt{free}(f(\vec{x}; \vec{y})) = \{\vec{x}, \vec{y}\}$ and $\texttt{mod}(f(\vec{x}; \vec{y})) = \{\vec{y}\}$.

For a given program, we assume that we have *small specifications* for all procedures and loop invariants for while loops in the program. These may be obtained from an interprocedural shape analysis based on separation logic, such as that described in [4], or could be given as annotations by hand, such as when doing a safety verification [2]. A specification for a procedure f is given by a spec table $\mathcal{T}(f) : \texttt{SH} \rightharpoonup \mathcal{P}(\texttt{SH})$, which is a partial function from symbolic heaps to sets of symbolic heaps. The intended meaning of $\mathcal{T}(f)$ is the set of Hoare triples $\{P\}\ f(\vec{x})\ \{\bigvee_{Q \in \mathcal{T}(f)} Q\}$ for every $P \in \texttt{SH}$ on which $\mathcal{T}(f)$ is defined. For a while loop $i : \texttt{while}\ b\ c$, the loop invariant is given as a set of symbolic heaps, the intended formula being the disjunction of all the symbolic heaps in this set. We assume that all symbolic heaps in procedure summaries and invariants have all empty labels.

Given these specifications and invariants for procedures and while loops, for our analysis we shall consider an intermediate language for commands in which procedure calls and while loops are replaced by *specified* commands, $\texttt{com}[\mathcal{T}]$, where $\mathcal{T}$ is a spec table.

$$c ::= i : A \mid i : \texttt{com}[\mathcal{T}] \mid i : \texttt{if}\ b\ c_1\ c_2 \mid c_1; c_2$$

A $\texttt{com}[\mathcal{T}]$ command is some command which satisfies the specification given by $\mathcal{T}$. For a while loop $\texttt{while}\ b\ c$ with invariant S, the spec table is obtained as a partial function that is only defined on symbolic heaps $\Pi \mid \Sigma \in S$, and maps each of these inputs to the set $\{\neg b \wedge \Pi \mid \Sigma \mid \Pi \mid \Sigma \in S\}$. Atomic and specified commands may be referred to as *basic* commands, and may be denoted by $i : B$. For any command c we let $I_b(c)$ be the set of indices of all basic commands in c.

4 Independence Detection

In this section we describe the algorithm for determining when two statements in a given program are independent in the sense that they do not access a common heap location

in any possible execution. The basic idea is to perform a symbolic execution [2] with labelled symbolic heaps, in which the labels keep track of regions of memory through the execution. The symbolic *footprint* of every program statement is recorded as the set of labels which represent the memory regions that are accessed in the execution of that statement. In order to determine independences between footprints, an *intersection* relation between labels needs to be maintained, which relates any two labels that represent possibly overlapping regions of memory.

Formally, we define a symbolic state as a triple $(\Pi \mid \Sigma, \mathcal{F}, \mathcal{I})$, where $\Pi \mid \Sigma$ is a labelled symbolic heap, $\mathcal{F}$ is a **footprint log**, and $\mathcal{I}$ is an **intersection log**. The footprint log is as a partial function $\mathcal{F} : I \rightharpoonup \mathcal{P}(\mathtt{Lab})$ which maps indices of commands to sets of labels which represent their footprint, and is updated for every command index when the command is encountered during symbolic execution. The intersection log $\mathcal{I} \in \mathcal{P}(\mathcal{P}_2(\mathtt{Lab}))$ (where $\mathcal{P}_2(\mathtt{Lab})$ is the set of all unordered pairs of labels) maintains a relation between labels that represent possibly overlapping regions of the heap.

4.1 Symbolic Execution Rules

Symbolic execution is based on a set of *operational* and *rearrangement* rules which determine the transformation of the symbolic states through the execution. The rules are displayed in figure 1, where they should be read from top to bottom, and they employ some expressions which we define below. The operational rules describe, for each kind of command, the effect of the command on the symbolic heap on which it executes safely. The footprint log is updated for the index of the command with the labels of the accessed portion of the symbolic heap, and the intersection log is updated when fresh labels are introduced that may possibly intersect with old ones. The first four rules are those for the atomic commands, where the footprint log is updated with the label of the accessed cell. The rules for lookup and mutation use the following definitions:

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

In the case of allocation, a fresh label is introduced for the newly allocated cell, but the intersection log is unchanged as the new label does not intersect with any old ones.

The last operational rule is for the specified commands. In this case the spec table, $\mathcal{T}$, determines the transformation of the symbolic heap. However, the assertion at the call-site may be larger than the pre-condition, since the pre-condition only describes the part of the heap that is accessed by the command. For this reason, the *frame assertion* needs to be discovered, which is the part of the call-site heap that is not in the pre-condition of the command. We describe the frame inference method in detail in section 6. For now, we use the expression $\mathit{frame}(\Pi \mid \Sigma, \Pi_1 \mid \Sigma_1)$ to denote the frame assertion obtained for call-site assertion $\Pi \mid \Sigma$ and pre-condition $\Pi_1 \mid \Sigma_1$. The transformed symbolic heap is obtained by the conjunction of the frame assertion (which preserves its labels from the call-site assertion) with a post-condition of the command, the empty labels of which are replaced by fresh non-empty ones: the expression $\mathit{freshlabs}(\Sigma_2, \Sigma_2')$ means that Σ_2' is the formula Σ_2 with fresh non-empty labels on all simple conjuncts.

As an example, consider the case where the call-site state is $(\langle x \mapsto [l : y, r : z] \rangle_1 * \langle \mathtt{tree}(y) \rangle_2 * \langle \mathtt{tree}(z) \rangle_3, \mathcal{F}, \mathcal{I})$ and the specified command is a call to a procedure

which rotates a tree at y, having a spec table with pre- and post- condition $\langle \texttt{tree}(y) \rangle_\bullet$. In this case the inferred frame assertion is $\langle x \mapsto [l : y, r : z] \rangle_1 * \langle \texttt{tree}(z) \rangle_3$. The fresh label 4 may be assigned to the post-condition, giving the transformed symbolic heap to be $\langle x \mapsto [l : y, r : z] \rangle_1 * \langle \texttt{tree}(y) \rangle_4 * \langle \texttt{tree}(z) \rangle_3$.

The footprint labels of the specified command are determined by the labels of the pre-condition and the post-condition heap. In the example, the footprint of the procedure call will be $\{2, 4\}$. Since fresh labels are introduced in the post-condition, the intersection log should be updated with the information of which labels the new labels may possibly intersect with. In the example, 4 possibly intersects with 2 and everything that 2 was already possibly intersecting with in $\mathcal{I}$. In the rule, we use the expression $relFresh$ to relate the fresh labels to all the labels in the pre-condition in this way. For a set of labels L we define $getIntLabels(L, \mathcal{I}) = \{l' \mid \{l, l'\} \in \mathcal{I} \wedge l \in L\}$. This is the set of labels that represent locations that may overlap with locations represented by labels in L, according to the intersection log $\mathcal{I}$. For sets of labels L_1 and L_2, we define

$$relFresh(L_1, L_2, \mathcal{I}) = \mathcal{I} \cup \{\{l_1, l_2\} \mid l_1 \in L_1 \wedge l_2 \in getIntLabels(L_2, \mathcal{I})\}$$

This expression is used to update the intersection log $\mathcal{I}$ when a fresh set of labels L_1 is introduced in such a way that any label in L_1 may intersect with any of the labels in L_2.

The rearrangement rules are needed to make an expression E explicit in the symbolic heap so that an operational rule for a command that accesses the heap cell at E can be applied. Apart from the first simple substitution rule, they are basically unfolding rules for each of the inductively defined data structure predicates, where fresh labels in the unfolding are related to the original using $relFresh$.

4.2 Independence Detection Algorithm

The independence detection algorithm is given in Figure 2. Given a command c with a set of preconditions Pre, the $getInd(c, Pre)$ method returns a set $Ind \subseteq \mathcal{P}_2(I_b(c))$ such that $\{i, j\} \in Ind$ implies that the basic statements with indices i and j are independent. For a conditional $i : \texttt{if}\ b\ c_1\ c_2$, we can test independence with a statement $j : c$ by testing independence between $j : c$ and all the basic statements in the conditional. The $track(S, c)$ method takes a command c and a set S of initial symbolic states, applies the execution rules from Figure 1, and returns the set of all possible output symbolic states. The footprint and intersection logs from all of these states are used by the $getInd$ method to find the independences. Once we have detected heap independences, we can use the `free` and `mod` sets of commands to determine stack independences, and then apply standard parallelization techniques such as those discussed in [7, 10].

5 Examples

We begin by illustrating our algorithm on a tree rotation program which is based on the main example from [10]. We have the procedure $rotateTree(x;)\{local\ x_1, x_2; c\}$, where the body c is shown in figure 3. The procedure takes a tree at x and rotates it by recursively swapping its left and right subtrees. Given the spec table with a single

OPERATIONAL RULES

$$\frac{(\Pi \mid \Sigma, \mathcal{F}, \mathcal{I})}{(x = E[x'/x] \wedge (\Pi \mid \Sigma)[x'/x], \mathcal{F}[i \to \emptyset], \mathcal{I})} \; i : x := E, x' fresh$$

$$\frac{(\Pi \mid \Sigma * \langle E \mapsto [\rho] \rangle_l, \mathcal{F}, \mathcal{I})}{(x = F[x'/x] \wedge (\Pi \mid \Sigma * \langle E \mapsto [\rho] \rangle_l)[x'/x], \mathcal{F}[i \to \{l\}], \mathcal{I})} \; i : x := E \to f, x' \; fresh, lookup(\rho, f) = F$$

$$\frac{(\Pi \mid \Sigma * \langle E \mapsto [\rho] \rangle_l, \mathcal{F}, \mathcal{I})}{(\Pi \mid \Sigma * \langle E \mapsto [\rho'] \rangle_l, \mathcal{F}[i \to \{l\}], \mathcal{I})} \; i : E \to f := F, mutate(\rho, f, F) = \rho'$$

$$\frac{(\Pi \mid \Sigma, \mathcal{F}, \mathcal{I})}{((\Pi \mid \Sigma)[x'/x] * \langle x \mapsto [] \rangle_l, \mathcal{F}[i \to \{l\}], \mathcal{I})} \; i : \texttt{new}(x), x' \; fresh, l \; fresh$$

$$\frac{(\Pi \mid \Sigma, \mathcal{F}, \mathcal{I})}{(\Pi \wedge \Pi_2 \mid \Sigma_2' * \Sigma_F, \mathcal{F}[i \to \texttt{Lab}(\Sigma_2') \cup (\texttt{Lab}(\Sigma) \backslash \texttt{Lab}(\Sigma_F))], relFresh(\texttt{Lab}(\Sigma_2'), \texttt{Lab}(\Sigma) \backslash \texttt{Lab}(\Sigma_F), \mathcal{I}))} \; \dagger$$

$$\dagger \; i : \texttt{com}[\mathcal{T}], \Pi_2 \mid \Sigma_2 \in \mathcal{T}(\Pi_1 \mid \Sigma_1), \Sigma_F = frame(\Pi \mid \Sigma, \Pi_1 \mid \Sigma_1), freshlabs(\Sigma_2, \Sigma_2')$$

REARRANGEMENT RULES

$$\frac{(\Pi \mid \Sigma * \langle F \mapsto [\rho] \rangle_l, \mathcal{F}, \mathcal{I})}{(\Pi \mid \Sigma * \langle E \mapsto [\rho] \rangle_l, \mathcal{F}, \mathcal{I})} \; \Pi \vdash E = F$$

$$\frac{(\Pi \mid \Sigma * \langle \texttt{ls}(F, F') \rangle_l, \mathcal{F}, \mathcal{I})}{(\Pi \mid \Sigma * \langle E \mapsto [n : x'] \rangle_{l_1} * \langle \texttt{ls}(x', F') \rangle_{l_2}, \mathcal{F}, relFresh(\{l_1, l_2\}, \{l\}, \mathcal{I}))} \; \dagger$$

$$\dagger \; \Pi \mid \Sigma * \texttt{ls}(F, F') \vdash F \neq F' \wedge E = F \text{ and } x' \text{ fresh and } l_1, l_2 \text{ fresh}$$

$$\frac{(\Pi \mid \Sigma * \langle \texttt{dlls}(F, F_b, F', F_b') \rangle_l, \mathcal{F}, \mathcal{I})}{(\Pi \mid \Sigma * \langle E \mapsto [n : x', b : F_b] \rangle_{l_1} * \langle \texttt{dlls}(x', E, F', F_b') \rangle_{l_2}, \mathcal{F}, relFresh(\{l_1, l_2\}, \{l\}, \mathcal{I}))} \; \dagger$$

$$\dagger \; \Pi \mid \Sigma * \texttt{dlls}(F, F_b, F', F_b') \vdash F \neq F' \wedge E = F \text{ and } x' \text{ fresh and } l_1, l_2 \text{ fresh}$$

$$\frac{(\Pi \mid \Sigma * \langle \texttt{dlls}(F, F_b, F', F_b') \rangle_l, \mathcal{F}, \mathcal{I})}{(\Pi \mid \Sigma * \langle \texttt{dlls}(F, F_b, E, x') \rangle_{l_1} * \langle E \mapsto [n : F', b : x'] \rangle_{l_2}, \mathcal{F}, relFresh(\{l_1, l_2\}, \{l\}, \mathcal{I}))} \; \dagger$$

$$\dagger \; \Pi \mid \Sigma * \texttt{dlls}(F, F_b, F', F_b') \vdash F \neq F' \wedge E = F_b' \text{ and } x' \text{ fresh and } l_1, l_2 \text{ fresh}$$

$$\frac{(\Pi \mid \Sigma * \langle \texttt{tree}(F) \rangle_l, \mathcal{F}, \mathcal{I})}{(\Pi \mid \Sigma * \langle E \mapsto [l : x', r : y'] \rangle_{l_1} * \langle \texttt{tree}(x') \rangle_{l_2} * \langle \texttt{tree}(y') \rangle_{l_3}, \mathcal{F}, relFresh(\{l_1, l_2, l_3\}, \{l\}, \mathcal{I}))} \; \dagger$$

$$\dagger \; \Pi \mid \Sigma * \texttt{tree}(F) \vdash F \neq \texttt{nil} \wedge E = F \text{ and } x', y' \text{ fresh and } l_1, l_2, l_3 \text{ fresh}$$

Fig. 1. Rules for symbolic execution with footprint tracking

```
getInd(c, Pre) =
  S := ∅
  for all Π¦Σ ∈ Pre
    assign fresh non-empty labels in Π¦Σ
    F := ∅
    I := {{l, l} | l ∈ Lab(Σ)}
    S := S ∪ track({(Π¦Σ, F, I)}, c)
  Ind := {i, j | i, j ∈ I_b(c)}
  for all i, j ∈ I_b(c)
    for all (Π¦Σ, F, I) ∈ S
      if there exist l ∈ F(i) and k ∈ F(j) such that
        {l, k} ∈ I then remove {(i, j)} from Ind
  return Ind
```

```
track(S, c) =
  if c is empty then return S
  else let c = i : c'; c''
    S' := ∅
    for all (Π¦Σ, F, I) ∈ S
      if c' is atomic command A and (Π¦Σ, F, I) matches premise
        of operational rule for A then add the conclusion to S'
      elseif c' is atomic command A accessing heap cell E and (Π¦Σ, F, I)
        matches premise of a rearrangment rule for E then add the conclusion to S'
      elseif c' = com[T] then
        for all P ∈ T for which frame inference succeeds
          for all Q ∈ T(P)
            add the conclusions of operational rule for com[T] to S'
      elseif c' = if b c1 c2 then
        S1 := track((b ∧ Π¦Σ, F, I), c1)
        S2 := track((¬b ∧ Π¦Σ, F, I), c2)
        S' := S' ∪ S1 ∪ S2
      else return fail
    return track(S', c'')
```

Fig. 2. Independence Detection Algorithm

pre-condition $\langle \texttt{tree}(x) \rangle_\bullet$ and single post-condition $\langle \texttt{tree}(x) \rangle_\bullet$, the execution of the independence detection algorithm is shown in figure 3. At the end of the execution, for final footprint log $\mathcal{F}_6$, we have $\mathcal{F}_6(i_6) = \{3, 5\}$ and $\mathcal{F}_6(i_7) = \{4, 6\}$. Since these labels do not intersect according to the final intersection log $\mathcal{I}_3$, we have that the two recursive calls i_6 and i_7 are independent, and therefore may be executed in parallel. Similar examples are given by other divide-and-conquer programs, such as a program for copying a tree and mergesort on linked lists, in which our algorithm determines the recursive calls to be independent.

Previous approaches such as [7, 10] have been based on *reachability* properties of certain pointer data structures to detect independences, e.g., statements referring to the left and right subtrees of a tree can be determined to be independent since no heap location is reachable from both of them. The limitations of this approach can be seen even on simple list segment programs, where reachability analysis is unable to guarantee independence since the list segment may in fact be part of a larger cyclic data structure. Worse is the situation where there is internal sharing within the data structure, such as in the case of doubly linked lists. In contrast, our approach does not suffer from these inherent limitations since it is based on detecting the *footprints* of statements, that is, the cells that are actually accessed rather than all the ones that may possibly be accessed. We illustrate this with the example in figure 4. In this case we have the procedure $setBack(x, y, z;)\{local\ x_1; c\}$, which transforms a singly linked list segment from x to y into a doubly linked segment by recursively traversing the segment and setting the back pointers. The body c is shown in the figure. The parameter z is the back

$(\langle \texttt{tree(x)} \rangle_1,\ \emptyset,\ \emptyset)$

i_1 : if$(x \neq \texttt{nil})\{$

$(\texttt{x} \neq \texttt{nil} \,\vert\, \langle \texttt{tree(x)} \rangle_1,\ \emptyset,\ \emptyset)$

$(\texttt{x} \neq \texttt{nil} \,\vert\, \langle \texttt{x} \mapsto [\texttt{l} : \texttt{x}', \texttt{r} : \texttt{y}'] \rangle_2 * \langle \texttt{tree(x}') \rangle_3 * \langle \texttt{tree(y}') \rangle_4,\ \emptyset,\ \mathcal{I}_1)$

i_2 : $x_1 := x \rightarrow l;$

$(\texttt{x}_1 = \texttt{x}' \wedge \texttt{x} \neq \texttt{nil} \,\vert\, \langle \texttt{x} \mapsto [\texttt{l} : \texttt{x}', \texttt{r} : \texttt{y}'] \rangle_2 * \langle \texttt{tree(x}') \rangle_3 * \langle \texttt{tree(y}') \rangle_4,\ \mathcal{F}_1 = i_2 \rightarrow \{2\},\ \mathcal{I}_1)$

i_3 : $x_2 := x \rightarrow r;$

$(\texttt{x}_2 = \texttt{y}' \wedge \texttt{x}_1 = \texttt{x}' \wedge \texttt{x} \neq \texttt{nil} \,\vert\, \langle \texttt{x} \mapsto [\texttt{l} : \texttt{x}', \texttt{r} : \texttt{y}'] \rangle_2 * \langle \texttt{tree(x}') \rangle_3 * \langle \texttt{tree(y}') \rangle_4,\ \mathcal{F}_2 = \mathcal{F}_1[i_3 \rightarrow \{2\}],\ \mathcal{I}_1)$

i_4 : $x \rightarrow l := x_2;$

$(\texttt{x}_2 = \texttt{y}' \wedge \texttt{x}_1 = \texttt{x}' \wedge \texttt{x} \neq \texttt{nil} \,\vert\, \langle \texttt{x} \mapsto [\texttt{l} : \texttt{x}_2, \texttt{r} : \texttt{y}'] \rangle_2 * \langle \texttt{tree(x}') \rangle_3 * \langle \texttt{tree(y}') \rangle_4,\ \mathcal{F}_3 = \mathcal{F}_2[i_4 \rightarrow \{2\}],\ \mathcal{I}_1)$

i_5 : $x \rightarrow r := x_1;$

$(\texttt{x}_2 = \texttt{y}' \wedge \texttt{x}_1 = \texttt{x}' \wedge \texttt{x} \neq \texttt{nil} \,\vert\, \langle \texttt{x} \mapsto [\texttt{l} : \texttt{x}_2, \texttt{r} : \texttt{x}_1] \rangle_2 * \langle \texttt{tree(x}') \rangle_3 * \langle \texttt{tree(y}') \rangle_4,\ \mathcal{F}_4 = \mathcal{F}_3[i_5 \rightarrow \{2\}],\ \mathcal{I}_1)$

i_6 : $rotateTree(x_1;);$

$(\texttt{x}_2 = \texttt{y}' \wedge \texttt{x}_1 = \texttt{x}' \wedge \texttt{x} \neq \texttt{nil} \,\vert\, \langle \texttt{x} \mapsto [\texttt{l} : \texttt{x}_2, \texttt{r} : \texttt{x}_1] \rangle_2 * \langle \texttt{tree(x}_1) \rangle_5 * \langle \texttt{tree(y}') \rangle_4,\ \mathcal{F}_5 = \mathcal{F}_4[i_6 \rightarrow \{3, 5\}],\ \mathcal{I}_2)$

i_7 : $rotateTree(x_2;);$

$(\texttt{x}_2 = \texttt{y}' \wedge \texttt{x}_1 = \texttt{x}' \wedge \texttt{x} \neq \texttt{nil} \,\vert\, \langle \texttt{x} \mapsto [\texttt{l} : \texttt{x}_2, \texttt{r} : \texttt{x}_1] \rangle_2 * \langle \texttt{tree(x}_1) \rangle_5 * \langle \texttt{tree(x}_2) \rangle_6,\ \mathcal{F}_6 = \mathcal{F}_5[i_7 \rightarrow \{4, 6\}],\ \mathcal{I}_3)$

$\}$

where $\mathcal{I}_1 = \{\{1,2\},\{1,3\},\{1,4\}\}, \mathcal{I}_2 = \mathcal{I}_1 \cup \{\{5,3\},\{5,1\}\}, \mathcal{I}_3 = \mathcal{I}_2 \cup \{\{6,4\},\{6,1\}\}$

Fig. 3. Independence detection for $rotateTree$

pointer to be set for the head element. In this case we have the spec table with a single pre-condition $\langle \texttt{ls}(x, y) \rangle_\bullet$ and single post-condition $\langle \texttt{dlls}(x, z, y, z') \rangle_\bullet$, where z' is the existentially quantified pointer to the last element. As can be seen in figure 4, our algorithm detects the recursive call at i_4 to be independent of the statement i_3, and they can hence be executed in parallel. A reachability-based approach will fail to determine this independence even though the statements are accessing disjoint locations.

6 Frame Inference with Label Respecting Entailment

In the operational rule for specified commands, we described how the frame assertion needs to be inferred to match the pre-condition to the call-site assertion. For this we adapt the frame inference method of [2], which uses a proof theory for entailments between symbolic heaps. However, apart from the formula itself, in this case we also require that the inferred frame assertion should correctly preserve its labels from the original call-site assertion, as these are used to determine the footprint labels of the specified command. For this we require a stronger notion of entailment, which we call *label respecting* entailment.

The standard meaning of an entailment $\Pi_1 \,\vert\, \Sigma_1 \vdash \Pi_2 \,\vert\, \Sigma_2$ between two symbolic heaps is given as $\forall s, h.\ s, h \models \Pi_1 \,\vert\, \Sigma_1$ implies $s, h \models \Pi_2 \,\vert\, \Sigma_2$. For label respecting entailment, we have the additional constraint that a label appearing on both sides of the entailment refers to the same locations in the heap. The definition of this form of entailment is based on the following property of labelled symbolic heaps.

$(\langle \mathtt{ls(x,y)} \rangle_1,\ \emptyset,\ \emptyset)$

i_1 : if$(x \neq y)\{$

$(\mathtt{x} \neq \mathtt{y} \mid \langle \mathtt{ls(x,y)} \rangle_1,\ \emptyset,\ \emptyset)$

$(\mathtt{x} \neq \mathtt{y} \mid \langle \mathtt{x} \mapsto [\mathtt{n} : \mathtt{x'}] \rangle_2 * \langle \mathtt{ls(x',y)} \rangle_3,\ \emptyset,\ \{\{2,1\},\{3,1\}\})$

i_2 : $x_1 := x \to n;$

$(\mathtt{x}_1 = \mathtt{x'} \wedge \mathtt{x} \neq \mathtt{y} \mid \langle \mathtt{x} \mapsto [\mathtt{n} : \mathtt{x'}] \rangle_2 * \langle \mathtt{ls(x',y)} \rangle_3,\ \mathcal{F}_1 = i_2 \to \{2\},\ \{\{2,1\},\{3,1\}\})$

i_3 : $x \to b := z;$

$(\mathtt{x}_1 = \mathtt{x'} \wedge \mathtt{x} \neq \mathtt{y} \mid \langle \mathtt{x} \mapsto [\mathtt{n} : \mathtt{x'}, \mathtt{b} : \mathtt{z}] \rangle_2 * \langle \mathtt{ls(x',y)} \rangle_3,\ \mathcal{F}_2 = \mathcal{F}_1[i_3 \to \{2\}],\ \{\{2,1\},\{3,1\}\})$

i_4 : $setBack(x_1, y, x)$

$(\mathtt{x}_1 = \mathtt{x'} \wedge \mathtt{x} \neq \mathtt{y} \mid \langle \mathtt{x} \mapsto [\mathtt{n} : \mathtt{x'}, \mathtt{b} : \mathtt{z}] \rangle_2 * \langle \mathtt{dlls(x_1,x,y,z')} \rangle_4,\ \mathcal{F}_3 = \mathcal{F}_2[i_4 \to \{3,4\}],\ \{\{2,1\},\{3,1\},\{4,3\},\{4,1\}\})$

$\}$

Fig. 4. Independence detection for $setBack$

Lemma 1. *If $s, h \models \Pi \mid \Sigma * \langle S \rangle_l$ and $l \neq \bullet$, then there is a unique h' such that $h = h' * h''$ and $s, h' \models \Pi \mid \langle S \rangle_l$. In this case we define $subheap(s, h, \Pi \mid \Sigma * \langle S \rangle_l, l) = h'$, and it is undefined otherwise.*

Proof: *The result follows by the fact that every formula is precise [14], that is, for any heap, there is at most a unique subheap that satisfies the formula.* ■

Definition 1 (Label respecting entailment). *The entailment $\Pi_1 \mid \Sigma_1 \vdash \Pi_2 \mid \Sigma_2$ holds iff for all s, h, $s, h \models \Pi_1 \mid \Sigma_1$ implies $s, h \models \Pi_2 \mid \Sigma_2$, and if $l \in \mathtt{Lab}(\Sigma_1)$ and $l \in \mathtt{Lab}(\Sigma_2)$ and $l \neq \bullet$ then $subheap(s, h, \Pi_1 \mid \Sigma_1, l) = subheap(s, h, \Pi_2 \mid \Sigma_2, l)$.*

We have adapted the proof theory for entailments from [2] for label respecting entailment in figure 5. We omit the normalization rules and rules for the tree and doubly linked segment predicates as they adapt in a very similar manner. In the figure, the expression $\mathtt{op}(E)$ is an abbreviation for $E \mapsto [\rho]$, $\mathtt{ls}(E, F)$, $\mathtt{dlls}(E, E_b, F, F_b)$ or $\mathtt{tree}(E)$. The guard $\mathtt{G}(\mathtt{op}(E))$ asserts that the heap is non-empty, and is defined as

$$\mathtt{G}(E \mapsto [\rho]) \triangleq \mathtt{true} \qquad \mathtt{G}(\mathtt{ls}(E, F)) \triangleq E \neq F$$

$$\mathtt{G}(\mathtt{dlls}(E, E_b, F, F_b)) \triangleq E \neq F \wedge E_b \neq F_b \qquad \mathtt{G}(\mathtt{tree}(E)) \triangleq E \neq \mathtt{nil}$$

The label respecting aspect of these rules can be best appreciated by considering the way in which the frame inference method works. Assume we are given a call-site assertion $\Pi \mid \Sigma$ and procedure pre-condition $\Pi_1 \mid \Sigma_1$. We want to infer a frame Σ_F such that $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F$. This is done by applying the proof rules upwards from the entailment $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1$, as instructed by the following theorem which we inherit from [2] for label-respecting entailment.

Theorem 1. *Suppose that we have an incomplete proof:*

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

Then there is a complete proof of $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F$.

$$\frac{}{\Pi \mid \mathtt{emp} \vdash \mathtt{true} \mid \mathtt{emp}} \quad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \wedge E = E \mid \Sigma'}$$

$$\frac{\Pi \wedge P \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge P \mid \Sigma \vdash \Pi' \wedge P \mid \Sigma'} \quad \frac{\langle S \rangle_l \vdash \langle S' \rangle_k \quad \Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \langle S \rangle_l * \Sigma \vdash \Pi' \mid \langle S' \rangle_k * \Sigma'} \; l, k \in \{\bullet\} \cup \mathtt{Lab} \backslash (\mathtt{Lab}(\Sigma) \cup \mathtt{Lab}(\Sigma'))$$

$$\frac{}{\langle S \rangle_l \vdash \langle S \rangle_k} \quad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid \langle \mathtt{ls}(E, E) \rangle_l * \Sigma'} \; l \in \{\bullet\} \cup \mathtt{Lab} \backslash \mathtt{Lab}(\Sigma')$$

$$\frac{\Pi \wedge E_1 \neq E_3 \mid \langle E_1 \mapsto E_2 \rangle_{l_1} * \Sigma \;\vdash\; \Pi' \mid \langle E_1 \mapsto E_2 \rangle_{l_2} * \langle \mathtt{ls}(E_2, E_3) \rangle_{l_3} * \Sigma'}{\Pi \wedge E_1 \neq E_3 \mid \langle E_1 \mapsto E_2 \rangle_{l_1} * \Sigma \;\vdash\; \Pi' \mid \langle \mathtt{ls}(E_1, E_3) \rangle_{l_4} * \Sigma'} \; l_4 \in \{\bullet\} \cup \mathtt{Lab} \backslash (\mathtt{Lab}(\Sigma) \cup \mathtt{Lab}(\Sigma') \cup \{l_1, l_2, l_3\})$$

$$\frac{\Pi \mid \langle \mathtt{ls}(E_1, E_2) \rangle_{l_1} * \Sigma \;\vdash\; \Pi' \mid \langle \mathtt{ls}(E_1, E_2) \rangle_{l_2} * \langle \mathtt{ls}(E_2, \mathtt{nil}) \rangle_{l_3} * \Sigma'}{\Pi \mid \langle \mathtt{ls}(E_1, E_2) \rangle_{l_1} * \Sigma \;\vdash\; \Pi' \mid \langle \mathtt{ls}(E_1, \mathtt{nil}) \rangle_{l_4} * \Sigma'} \; l_4 \in \{\bullet\} \cup \mathtt{Lab} \backslash (\mathtt{Lab}(\Sigma) \cup \mathtt{Lab}(\Sigma') \cup \{l_1, l_2, l_3\})$$

$$\frac{\Pi \wedge \mathtt{G}(\mathtt{op}(E_3)) \mid \langle \mathtt{ls}(E_1, E_2) \rangle_{l_1} * \langle \mathtt{op}(E_3) \rangle_{l_2} * \Sigma \;\vdash\; \Pi' \mid \langle \mathtt{ls}(E_1, E_2) \rangle_{l_3} * \langle \mathtt{ls}(E_2, E_3) \rangle_{l_4} * \Sigma'}{\Pi \wedge \mathtt{G}(\mathtt{op}(E_3)) \mid \langle \mathtt{ls}(E_1, E_2) \rangle_{l_1} * \langle \mathtt{op}(E_3) \rangle_{l_2} * \Sigma \;\vdash\; \Pi' \mid \langle \mathtt{ls}(E_1, E_3) \rangle_{l_5} * \Sigma'} \; \dagger$$

$$\dagger \; l_5 \in \{\bullet\} \cup \mathtt{Lab} \backslash (\mathtt{Lab}(\Sigma) \cup \mathtt{Lab}(\Sigma') \cup \{l_1, l_2, l_3, l_4\})$$

Fig. 5. Rules for label respecting entailment

In our label respecting adaptation of the proof rules, when applying the rules upwards, labels can only be removed from the left hand side of an entailment. Hence the frame assertion Σ_F will retain its labels from the call-site assertion $\Pi \mid \Sigma$. By theorem 1, the entailment $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F$ is label respecting, and so we have that the labels common to the call-site assertion and the frame assertion refer to the same heap locations. Notice that when applying this method in practice, since we are only concerned about preserving the labels in the frame assertion, we do not care about the labels on the right hand side of the entailments as we go up the proof. They can hence all be chosen to be the empty label when applying the rules upwards. As a simple illustration, in the case where the call-site assertion is $\langle x \mapsto [l : y, r : z] \rangle_1 * \langle \mathtt{tree}(y) \rangle_2 * \langle \mathtt{tree}(z) \rangle_3$ and the procedure pre-condition is $\langle \mathtt{tree}(y) \rangle_\bullet$, the following one-step derivation gives us the correctly labelled frame assertion:

$$\frac{\langle x \mapsto [l : y, r : z] \rangle_1 * \langle \mathtt{tree}(z) \rangle_3 \vdash \mathtt{emp}}{\langle x \mapsto [l : y, r : z] \rangle_1 * \langle \mathtt{tree}(y) \rangle_2 * \langle \mathtt{tree}(z) \rangle_3 \vdash \langle \mathtt{tree}(y) \rangle_\bullet}$$

7 Soundness

We demonstrate the soundness of our algorithm in detecting independences, a property which is necessary if we are to use the algorithm to safely parallelize a program. For this we use a trace semantics of programs which is based on the one described in [6].

α	$[\![\alpha]\!](s,h), loc(\alpha,s,h)$
$x := E$	$\{s[x \mapsto [\![E]\!]s], h\}, \emptyset$
$x := E \to f$	$\begin{cases} \{s[x \mapsto v], h\}, \{l\} & \text{if } [\![E]\!]s = l, l \in \texttt{Loc} \text{ and } h(l)(f) = v \\ \top, \emptyset & \text{otherwise} \end{cases}$
$E_1 \to f := E_2$	$\begin{cases} \{s, h[l \mapsto r]\}, \{l\} & \text{if } [\![E_1]\!]s = l, [\![E_2]\!]s = v, l \in \texttt{Loc} \text{ and } r = h(l)[f \to v] \\ \top, \emptyset & \text{otherwise} \end{cases}$
$\texttt{new}_l(x)$	$\begin{cases} \{s, h * l \mapsto r\}, \{l\} & \text{if } l \in \texttt{Loc} \backslash dom(h) \text{ and } r(f) = \texttt{nil} \text{ for all } f \in \texttt{Fields} \\ \emptyset, \emptyset & \text{otherwise} \end{cases}$
$\texttt{assume}(b)$	$\begin{cases} \{s, h\}, \emptyset & \text{if } [\![b]\!]s \\ \emptyset, \emptyset & \text{otherwise} \end{cases}$

Fig. 6. Denotational semantics and location sets of primitive actions

7.1 Action trace semantics

The action traces are made up of primitive actions α,

$$\alpha ::= x := E \mid x := E \to f \mid E_1 \to f := E_2 \mid \texttt{new}_l(x) \mid \texttt{assume}(b) \quad \text{where } l \in Loc$$

The $\texttt{assume}(b)$ action is used to implement conditionals, as shown in the trace semantics of commands below. It filters out states which satisfy the boolean b. The $new_l(x)$ command allocates the location l if it is not already allocated. We choose this instead of a non-deterministic allocation primitive (which is usually used in separation logic works) as keeping traces deterministic will be useful for our purposes.

Semantically, the primitive actions correspond to total functions that are of the form $\texttt{Stacks} \times \texttt{Heaps} \to \mathcal{P}(\texttt{Stacks} \times \texttt{Heaps})^\top$. The $\top$ element represents a faulting execution, that is, dereferencing a null pointer or an unallocated region of the heap. For a primitive action α and a state $(s,h) \in \texttt{Stacks} \times \texttt{Heaps}$, we define the **location set** $loc(\alpha, s, h)$ as the set of locations that are accessed by α when executed on the state (s,h). The denotational semantics and location sets of the primitive actions is given in figure 6.

Definition 2 (Action trace). *An action trace τ is a finite sequential composition of atomic actions, $\tau ::= \alpha; \cdots ; \alpha$*

Denotational semantics of action traces is given by the sequential composition of actions, which is defined as

$$[\![\alpha_1; \alpha_2]\!](s,h) = \begin{cases} \bigcup_{(s',h') \in [\![\alpha_1]\!](s,h)} [\![\alpha_2]\!](s',h') & \text{if } [\![\alpha_1]\!](s,h) \neq \top \\ \top & \text{otherwise} \end{cases}$$

Note that every trace τ is deterministic in that for any state (s,h), $[\![\tau]\!](s,h)$ either faults or has at most a single outcome $\{(s',h')\}$.

The action trace semantics of commands of our programming language is given in figure 7. Just as our commands are indexed, we assign unique indices to the primitive actions in every action trace of every command as follows. For each atomic command $i : A$, every trace is a single primitive action α, and we index this as $(i,1) : \alpha$.

$$T(x := E) = \{x := E\} \qquad T(x := [E]) = \{x := [E]\}$$

$$T([E_1] := [E_2]) = \{[E_1] := [E_2]\} \qquad T(\texttt{new}(x)) = \{\texttt{new}_l(x) \mid l \in \texttt{Loc}\}$$

$$T(\texttt{com}(\mathcal{T})) \subseteq \{\tau \mid \forall P \in dom(\mathcal{T}).\forall (s,h) \in [\![P]\!].\exists Q \in \mathcal{T}(P).\ [\![\tau]\!](s,h) \subseteq [\![Q]\!]\}$$

$$T(c_1; c_2) = \{\tau_1; \tau_2 \mid \tau_1 \in T(c_1), \tau_2 \in T(c_2)\}$$

$$T(\texttt{if}\ b\ c_1\ c_2) = \{\texttt{assume}(b); \tau_1 \mid \tau_1 \in T(c_1)\} \cup \{\texttt{assume}(\neg b); \tau_2 \mid \tau_2 \in T(c_2)\}$$

Fig. 7. Action trace semantics of commands

For each specified command $i : \texttt{com}(\mathcal{T})$, every trace $\alpha_1; ...; \alpha_n$ is indexed as $(i,1) : \alpha_1; ...; (i,n) : \alpha_n$. For sequential composition the indices are obtained from the component commands. For a conditional $i : \texttt{if}\ b\ c_1\ c_2$, we index the assume actions as $(i,1) : \texttt{assume}(b)$ and $(i,1) : \texttt{assume}(\neg b)$ and the other indices are obtained from the component commands. We shall write $(i,j) : \alpha \in \tau$ to mean that $\tau = \tau'; (i,j) : \alpha; \tau''$ for some τ' and τ''.

Definition 3 (Index subtrace). *For a trace τ and a command index i, we define $\tau|_i$ to be the subtrace of τ containing all the actions of the form $(i,j) : \alpha$. If there are no such actions in τ then $\tau|_i$ is undefined.*

Lemma 2. *For a command c, every trace $\tau \in T(c)$ is of the form $\tau|_{i_1}; ...; \tau|_{i_n}$, where $i_1, ..., i_n \in I(c)$.*

We define the locations accessed by an atomic action in the execution of a trace.

Definition 4 (Location set of an action in a trace). *The location set of an action $(i,j) : \alpha$ in a trace τ from initial state (s,h) is defined as*

$$loc((i,j) : \alpha, \tau, s, h) = \begin{cases} loc(\alpha, s', h') & \text{if } \tau = \tau_1; (i,j) : \alpha; \tau_2 \text{ and } [\![\tau_1]\!](s,h) = \{(s',h')\} \\ \emptyset & \text{otherwise} \end{cases}$$

We extend the definition of locations accessed by an action to the locations accessed by a subtrace of τ.

Definition 5 (Location set of a subtrace). *The location set of subtrace τ' of τ from initial state (s,h) is defined as* $loc(\tau', \tau, s, h) = \bigcup_{(i,j):\alpha \in \tau'} loc((i,j) : \alpha, \tau, s, h)$

We now give the formal definition of independence between two basic commands in a progam for a given pre-condition.

Definition 6 (Independence). *Given a command c and a pre-condition given by a set of symbolic heaps Pre, for two basic commands with indices i and i' in c, we say that command i is* **independent** *of command i', written $indep(i, i', c, Pre)$, iff for all $\Pi \mid \Sigma \in Pre$ and for all $(s,h) \in [\![\Pi \mid \Sigma]\!]$, we have for every $\tau \in T(c)$ such that $\tau|_i$ and $\tau|_{i'}$ are defined, that $loc(\tau|_i, \tau, s, h) \cap loc(\tau|_{i'}, \tau, s, h) = \emptyset$.*

7.2 Proof of soundness

Given the trace model described above, we can now formally state and prove the soundness property of the independence detection algorithm.

Theorem 2. *For a command c and a pre-condition set Pre, if for two basic commands with indices i and i' in c we have $\{i, i'\} \in getInd(c, Pre)$, then $indep(i, i', c, Pre)$.*

The algorithm of figure 2 works by applying the operational and rearrangement rules of figure 1 through the program, possibly branching on disjunctive outcomes and conditionals. We can therefore think of it as determining a set of *symbolic execution traces*, which are sequences of symbolic states, each starting with some initial state ψ_I given by the pre-condition and ending with some ψ_F in the final set of symbolic states that is used to determine independences.

Before we define symbolic execution traces, we formulate the case for conditionals in terms of an operational rule for assume statements. When the algorithm encounters a conditional statement with guard b, it branches on the two cases b and $\neg b$. Given the semantics of conditionals described in the last section, any action trace of the program at this point either starts with an $\texttt{assume}(b)$ or an $\texttt{assume}(\neg b)$ action. This step can hence be interpreted with an operational rule for assume statements:

$$\frac{(\Pi \mid \Sigma, \mathcal{F}, \mathcal{I})}{(b \wedge \Pi \mid \Sigma, \mathcal{F}, \mathcal{I})}\ i : \texttt{assume}(b)$$

Definition 7 (Symbolic execution trace). *A* **symbolic execution trace** $\mathcal{S}$ *is a sequence of symbolic states such that any two consecutive states in the sequence are related by an application of an operational or a rearrangement rule. The initial state is denoted $\mathcal{S}[0, 0]$, its symbolic heap has all non-empty labels, its footprint log is $\emptyset$ and its intersection log is $\{\{l, l\} \mid l \in \texttt{Lab}(\Pi \mid \Sigma)\}$, where $\Pi \mid \Sigma$ is the symbolic heap of $\mathcal{S}[0, 0]$.*

Apart from the initial state, every state is either an **operational state** *(the conclusion of an operational rule) or a* **rearrangement state** *(the conclusion of a rearrangement rule). The operational states are denoted $\mathcal{S}[1, 0]$ to $\mathcal{S}[N(\mathcal{S}), 0]$ in the order in which they appear, where $N(\mathcal{S})$ is the number of operational states and $N(\mathcal{S}) > 0$. For $0 \leq n < N(\mathcal{S})$, the rearrangement states from $\mathcal{S}[n, 0]$ to the next operational state are denoted $\mathcal{S}[n, 1], ..., \mathcal{S}[n, R(\mathcal{S}, n)]$, where $R(\mathcal{S}, n)$ is the number of rearrangement states in this segment. There are no rearrangement states after the last operational state.*

For a symbolic execution trace $\mathcal{S}$ and $0 \leq n \leq N(\mathcal{S})$, $0 \leq r \leq R(\mathcal{S}, n)$, we shall denote by $\mathcal{H}_{\mathcal{S}[n,r]}$, $\mathcal{F}_{\mathcal{S}[n,r]}$ and $\mathcal{I}_{\mathcal{S}[n,r]}$ the symbolic heap, footprint log and intersection log in state $\mathcal{S}[n, r]$ respectively. We denote by $i_{n,\mathcal{S}}$ the index of the command in the nth operational rule in $\mathcal{S}$, and let $I(\mathcal{S})$ be the set of all command indices. We let $\texttt{Lab}(\mathcal{S})$ be the set of all labels occuring in all the symbolic heaps in $\mathcal{S}$.

Definition 8 (Trace satisfaction). *Given a symbolic execution trace $\mathcal{S}$, we say that an action trace τ* **satisfies** *$\mathcal{S}$, written $\tau \models \mathcal{S}$, iff $\tau = \tau|_{i_{1,\mathcal{S}}}; ...; \tau|_{i_{N(\mathcal{S}),\mathcal{S}}}$ and for all $(s, h) \in [\![\mathcal{H}_{\mathcal{S}[0,0]}]\!]$, for all $1 \leq n \leq N(\mathcal{S})$ we have $[\![\tau|_{i_{1,\mathcal{S}}}; ...; \tau|_{i_{n,\mathcal{S}}}]\!](s, h) \subseteq [\![\mathcal{H}_{\mathcal{S}[n,0]}]\!]$.*

Lemma 3. *For a command c and pre-condition set Pre, let ψ_I be a symbolic state with a symbolic heap from Pre and footprint and intersection logs initialised as in the $getInd(c, Pre)$ method in figure 2. For every such initial state ψ_I, the algorithm generates a collection of symbolic execution traces, each starting with ψ_I and ending with some Ψ_F in the final set of states that is used to test independence. We have that every $\tau \in T(c)$ satisfies at least one of these symbolic execution traces.*

Proof: Trace satisfaction depends only on the symbolic heap component of the states in a symbolic trace and not on the footprint or intersection logs. Thus soundness of standard symbolic execution [2] alone implies that the algorithm overapproximates all possible executions of the program starting from the given pre-condition. ■

Proposition 1. *Assume we have $\tau \models \mathcal{S}$. Let $\mathcal{F}_F$ and $\mathcal{I}_F$ be the footprint and intersection logs of the final state of $\mathcal{S}$. For any two distinct command indices $i, i' \in I(\mathcal{S})$, if for all labels $l \in \mathcal{F}_F(i)$ and $l' \in \mathcal{F}_F(i')$ we have $\{l, l'\} \notin \mathcal{I}_F$, then for all $(s, h) \in [\![\mathcal{H}_{\mathcal{S}[0,0]}]\!]$, we have $loc(\tau|_i, \tau, s, h) \cap loc(\tau|_{i'}, \tau, s, h) = \emptyset$.*

The proof of this proposition appears in the appendix (section 9). The underlying idea is that given an action trace τ satisfying a symbolic execution trace $\mathcal{S}$, and a concrete initial state (s, h), every label l in the symbolic execution trace represents a fixed set of heap locations throughout the entire concrete execution of τ, which we denote $labloc(l, \mathcal{S}, \tau, s, h)$. This expression is then used to reason about the heap locations represented by labels in the footprint and intersection logs, and to show that two subtraces with non-intersecting footprint labels access disjoint heap locations.

Lemma 3 and proposition 1 together give the proof of the soundness theorem 2, as follows. Assume we are given a program c, a pre-condition set Pre, and indices i and i' of two basic commands, and that $\{i, i'\} \in getInd(c, Pre)$. Hence in each of the final symbolic states generated by the algorithm, the footprint labels of i and i' do not intersect according to the intersection log. By lemma 3, every trace τ of the program satisfies some symbolic execution trace generated by the algorithm. Hence, by proposition 1, if $\tau|_i$ and $\tau|_{i'}$ are defined then they have disjoint location sets starting from any state in the pre-condition. Since this is true for all traces of c, by definition of independence (definition 6), we have $indep(i, i', c, Pre)$.

8 Conclusion and Future Work

In this work we have focussed on laying the foundations of our extended separation logic framework for independence detection and program parallelization. We plan to extend the mechanism we describe to the more complex data structures handled by separation logic shape analyses, to integrate our method with the existing *space invader* tool for shape analysis [16,4], and conduct practical experiments, conceivably exploiting the scalability of this tool to large programs.

A notable aspect of this integration is that, while our framework relies on the atomic predicates being precise, sometimes imprecise predicates, e.g. 'possibly cyclic list', are used in shape analyses. However, these predicates are 'boundedly imprecise', so that

case analysis can be performed to obtain finite disjunctions of precise predicates from imprecise ones.

Another direction for future work is to improve the precision of label tracking by incorporating it into the shape analysis phase itself, which would involve taking the footprint and intersection logs through the abstraction and fixpoint calculations. This will also allow the detection of read/read independence by tracking the read-only labels in the specifications of while loops and procedures. Following this, we intend to investigate the application of our method to the various other kinds of program optimizations.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. OHearn, T. Wies and H. Yang. Shape Analysis for Composite Data Structures. In *CAV*, 2007.
2. J. Berdine, C. Calcagno, and P.W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, 2005.
3. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, 2006.
4. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis. To appear in *POPL*, 2009.
5. D. Distefano, P. O'Hearn, and H. Yang. A Local Shape Analysis based on Separation Logic. In *TACAS*, 2006.
6. C. Calcagno, P. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic. In *LICS*, 2007.
7. R. Ghiya, L. J. Hendren and Y. Zhu. Detecting Parallelism in C programs with recursive data structures. In *CC*, 1998.
8. B. Guo, N. Vachharajani, and D. August. Shape Analysis with Inductive Recursion Synthesis. In *PLDI*, 2007.
9. R. Gupta, S. Pande, K. Psarris and V. Sarkar. Compilation Techniques for Parallel Systems. In *Parallel Computing*, 1999.
10. L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. In *IEEE Transactions on Parallel and Distributed Systems*, 1990.
11. J. Hummel, L. J. Hendren and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *PLDI*, 1994.
12. T. Hoare and P. O'Hearn. Separation Logic Semantics of Communicating Processes. In *FICS*, 2008.
13. S. Horwitz, P. Pfeiffer and T. W. Reps. Dependence analysis for poiner variables. In *PLDI*, 1989.
14. P. OHearn, H. Yang, J. Reynolds. Separation and information hiding. In *POPL*, 2004.
15. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, 2002.
16. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable Shape Analysis for Systems Code. In *CAV*, 2008.