Automatic Parallelization with Separation Logic

Mohammad Raza, Cristiano Calcagno, and Philippa Gardner

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2AZ, UK
{mraza,ccris,pg}@doc.ic.ac.uk

Abstract. Separation logic is a recent approach to the analysis of pointer programs in which resource separation is expressed with a logical connective in assertions that describe the state at any given point in the program. We extend this approach to express properties of memory separation between *different* points in the program, and present an algorithm for determining independences between program statements 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 [8,9,12]. 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 [17,4,1]. 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 extend the separation logic approach to express memory separation properties throughout a program's lifetime.

The basic idea is to extend 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

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 348–362, 2009.

by a frame inference method [2] that keeps track of the labels by using a form of *label respecting* entailment between formulae.

Our approach fits in the line of work of using static analysis to detect independent statements in programs that manipulate pointer data structures [9,7,10,12,13]. Our departure point is the use of separation logic-based shape analysis. A logic-based approach is also advocated in [10], where *aliasing axioms* and theorem proving are used to detect independence. However, this method has difficulty handling structural modifications to the data structure, which do not cause problems in our case. Our method also does not rely on *reachability* properties of data structures, as in [9]. Such approaches encounter difficulties with data structure 'segments', such as non-nil-terminated list segments, and the situation is even worse when there is internal sharing within the data structure, as in the case of doubly linked lists. Our approach does not suffer from these inherent limitations as 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 on a program that converts a singly linked list segment into a doubly linked segment. A somewhat different approach to parallelization is proposed in [16], where *commutativity analysis* is used for identifying operations that produce the same output regardless of the order of execution. This method works together with an independence analysis, and works better depending on the strength of the independence analysis, and it will therefore be interesting to explore its combination with our method in future work.

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 $\mathtt{Heaps} = \mathtt{Loc} \rightharpoonup_{fin} (\mathtt{Fields} \to \mathtt{Val} \cup \mathtt{Loc})$ 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 spatial assertions 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

$E, F ::= \mathtt{nil} \mid x \mid x'$	expressions
$\rho ::= f_1 : E_1, ..., f_k : E_k$	record expressions
$\Pi ::= \mathtt{true} \mid E = E \mid E \neq E \mid \Pi \wedge \Pi$	pure assertions
$S ::= E \mapsto [\rho] \mid \mathtt{ls}(E, F) \mid \mathtt{dls}(E_f, E_b, F_f, F_b) \mid \mathtt{tree}(E)$	simple spatial assertions
$\Sigma ::= \mathtt{emp} \mid \langle S \rangle_l \mid \Sigma * \Sigma$	labelled spatial assertions
$\mathtt{SH} ::= \Pi \mid \Sigma$	symbolic heaps

The simple spatial assertions we consider in this paper are for list segments, doubly linked list segments and binary trees, the formal semantics of which are given below. Every simple spatial assertion (conjunct) in a symbolic heap has a label, which shall be used to keep track of the part of the heap that the conjunct is describing. The *empty* label $\bullet \in \mathtt{Lab}$ shall be used in situations where the label is unspecified. Except for the empty label, we require that every label has at most a unique occurrence in a symbolic heap. We let $L(\Pi \mid \Sigma)$ denote the set of labels in the symbolic heap $\Pi \mid \Sigma$.

Labels shall be interpreted in the context of a symbolic execution rather than on a single symbolic heap. This is because they shall be used to relate the states at different points through the execution of a program, and thus do not hold meaning on an individual state. The interpretation of symbolic heaps is therefore the standard one (ignoring the labels), 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. We write $h = h_0 * h_1$ to indicate 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.

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

$s, h \models E_1 = E_2$	iff	$[\![E_1]\!]s = [\![E_2]\!]s$
$s, h \models E_1 \neq E_2$	iff	$[\![E_1]\!]s \neq [\![E_2]\!]s$
$s, h \models \mathtt{true}$		always
$s, h \models \Pi_0 \wedge \Pi_1$	iff	$s, h \models \Pi_0$ and $s, h \models \Pi_1$
$s, h \models \langle E_0 \mapsto [f_1 : E_1, ..., f_k : E_k] \rangle_l$	iff	$h = [[\![E_0]\!]s \rightarrow r]$ where $r(f_i) = [\![E_i]\!]s$ for $i \in 1..k$
$s, h \models \langle \mathtt{ls}(E, F) \rangle_l$	iff	there is a linked list segment from E to F
$s, h \models \langle \mathtt{dls}(E_f, E_b, F_f, F_b) \rangle_l$	iff	there is a doubly linked list segment from E_f to F_f with initial and final back pointers E_b and F_b
$s, h \models \langle \mathtt{tree}(E) \rangle_l$	iff	there is a tree at E
$s, h \models \mathtt{emp}$	iff	$h = \emptyset$
$s, h \models \Sigma_0 * \Sigma_1$	iff	$\exists h_0 h_1 . h = h_0 * h_1$ and $s, h_0 \models \Sigma_0$ and $s, h_1 \models \Sigma_1$
$s, h \models \Pi \mid \Sigma$	iff	$\exists v . s(x' \mapsto v), h \models \Pi$ and $s(x' \mapsto v), h \models \Sigma$ where x' is the collection of primed variables in $\Pi \mid \Sigma$

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

$$\begin{aligned}
\texttt{ls}(E,F) &\Leftrightarrow (E = F \wedge \texttt{emp}) \vee (E \neq F \wedge \exists y. E \mapsto [n : y] * \texttt{ls}(y, F)) \\
\texttt{dls}(E_f,E_b,F_f,F_b) &\Leftrightarrow (E_f = F_f \wedge E_b = F_b \wedge \texttt{emp}) \vee \\
&\quad (E_f \neq F_f \wedge E_b \neq F_b \wedge \exists y. E_f \mapsto [n : y, b : E_b] * \texttt{dls}(y, E_f, F_f, F_b)) \\
\texttt{tree}(E) &\Leftrightarrow (E = \texttt{nil} \wedge \texttt{emp}) \vee (\exists x, y. E \mapsto [l : x, r : y] * \texttt{tree}(x) * \texttt{tree}(y))
\end{aligned}$$

3 Programming Language

We consider a standard programming language with procedures.

$b ::= E = E \mid E \neq E$	boolean expressions
$A ::= x := E \mid x := E \rightarrow f \mid E_1 \rightarrow f := E_2 \mid \texttt{new}(x)$	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 modified in the body, and $\vec{y} = y_1, .., y_m$ are the variables that are. We assume that all variables occurring 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 separation logic specifications for the procedure calls and loop invariants for the while loops. 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 [3]. Formally, a specification is represented by a *spec table*, $\mathcal{T} : \texttt{SH} \rightharpoonup \mathcal{P}(\texttt{SH})$, which is a partial function from symbolic heaps to sets of symbolic heaps. A spec table $\mathcal{T}$ for a command represents the set of Hoare triples in which, for every $P \in dom(\mathcal{T})$, there is a triple with pre-condition P and post-condition $\bigvee_{Q \in \mathcal{T}(P)} Q$. In the case of while loops, the loop invariant may be given as a set of symbolic heaps, the intended formula being the disjunction of all the symbolic heaps in this set. For a while loop $\texttt{while}\ b\ c$ with invariant S, we obtain the spec table as the partial function that is only defined on symbolic heaps $\Pi \mid \Sigma \in S$, and maps each of these inputs to the set $\{\neg b \wedge \Pi \mid \Sigma \mid \Pi \mid \Sigma \in S\}$. Given these specifications, 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 $\mathcal{T}$. We assume that all symbolic heaps in the spec tables of specified commands have empty labels. 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 $\mathcal{I} \in \mathcal{P}(\mathcal{P}_2(\mathtt{Lab}))$ is a set of unordered pairs of labels which determines 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 mutation and lookup use the following definitions:

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

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 pre- and post- conditions in the command's spec table determine the transformation of the symbolic heap. However, the assertion at the call-site may be larger than the command 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 $frame(\Pi \mid \Sigma, \Pi_1 \mid \Sigma_1)$ to denote the frame assertion obtained for call-site

assertion $\Pi \restriction \Sigma$ and pre-condition $\Pi_1 \restriction \Sigma_1$. The transformed symbolic heap is obtained by the conjunction of the frame assertion with the post-condition. The frame inference method ensures that the frame assertion preserves its labels from the call-site assertion. The post-condition assertion, which has all empty labels in the spec table, is assigned fresh non-empty labels with the expression $\mathit{freshlabs}(\Sigma_2, \Sigma_2')$, which 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 $\langle \mathtt{tree}(y)\rangle_\bullet$. In this case the inferred frame assertion is $\langle x \mapsto [l : y, r : z]\rangle_1 * \langle \mathtt{tree}(z)\rangle_3$. The fresh label 4 may be assigned to the post-condition, giving the transformed symbolic heap to be $\langle x \mapsto [l : y, r : z]\rangle_1 * \langle \mathtt{tree}(y)\rangle_4 * \langle \mathtt{tree}(z)\rangle_3$.

The footprint labels of the specified command are determined by the labels of the pre- and post- condition assertions. 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 rule, we use the expression $\mathit{relFresh}(L_1, L_2, \mathcal{I})$ 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 possibly intersect with any label in the set L_2, or with any label that intersects with some label in L_2 according to $\mathcal{I}$.

$$\mathit{relFresh}(L_1, L_2, \mathcal{I}) = \mathcal{I} \cup \{\{l_1, l\} \mid l_1 \in L_1 \wedge (l \in L_2 \vee \exists l' \in L_2.\ \{l, l'\} \in \mathcal{I})\}$$

In our example, if $\mathcal{I} = \{\{1,5\},\{2,5\},\{3,5\}\}$ then the transformed intersection log is given by $\mathit{relFresh}(\{4\},\{2\},\mathcal{I}) = \{\{1,5\},\{2,5\},\{3,5\},\{4,2\},\{4,5\}\}$, meaning that the fresh label 4 possibly intersects with 2 and everything that 2 was already possibly intersecting with in $\mathcal{I}$. Note that this example shows that the relation determined by the intersection log is not transitive. The intended relation is of course reflexive and symmetric, and this is taken into account in the independence detection algorithm.

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, these are basically unfolding rules for each of the inductively defined data structure predicates, where fresh labels in the unfolding are related to the original label using $\mathit{relFresh}$.

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 $\mathit{getInd}(c, \mathit{Pre})$ function returns a set $\mathit{Ind} \subseteq \mathcal{P}_2(I_b(c))$ such that $\{i, j\} \in \mathit{Ind}$ implies that the basic statements with indices i and j are independent. For a conditional $i : \mathtt{if}\ b\ c_1\ c_2$, we can test independence with a statement $j : c$ by testing independence between $j : c$ and all the basic statements in the conditional. The $\mathit{track}(S, c)$ function 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

OPERATIONAL RULES

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

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

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

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

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

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

REARRANGEMENT RULES

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

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

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

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

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

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

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

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

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

Fig. 1. Rules for symbolic execution with footprint tracking

states. The footprint and intersection logs from all of these states are used by the $getInd$ function 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 [7,9].

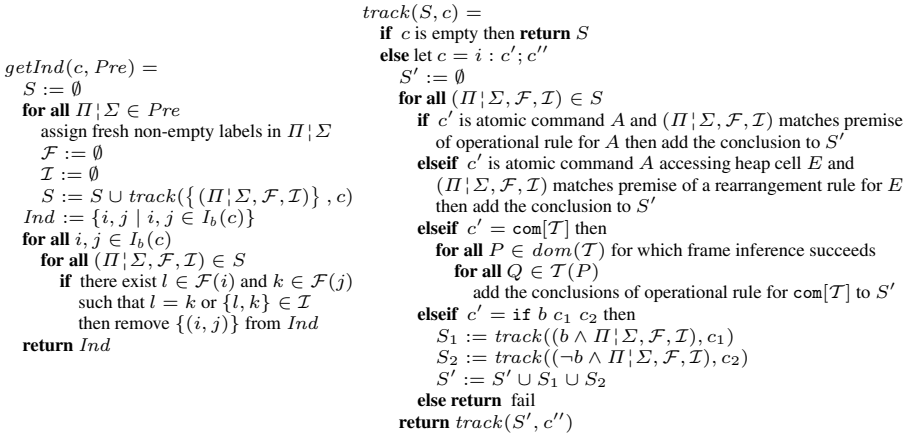

Fig. 2. Independence Detection Algorithm

5 Examples

We begin by illustrating our algorithm on a tree rotation program which is based on the main example from [9]. 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 pre-condition $\langle \mathtt{tree}(x) \rangle_\bullet$ and single post-condition $\langle \mathtt{tree}(x) \rangle_\bullet$, 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 $copyTree$ and $mergeSort$ on linked lists, in which our algorithm determines the recursive calls to be independent.

Previous approaches to independence detection such as [9] have been based on *reachability* properties of certain pointer data structures, 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

$$
\begin{array}{ll}
 & \left(\langle \mathtt{tree(x)}\rangle_1, \emptyset, \emptyset\right) \\
i_1: & \text{if}(x \neq \mathtt{nil})\{ \\
 & \left(\mathtt{x} \neq \mathtt{nil} \,\vert\, \langle \mathtt{tree(x)}\rangle_1, \emptyset, \emptyset\right) \\
 & \left(\mathtt{x} \neq \mathtt{nil} \,\vert\, \langle \mathtt{x} \mapsto [\mathtt{l}:\mathtt{x}', \mathtt{r}:\mathtt{y}']\rangle_2 * \langle \mathtt{tree(x')}\rangle_3 * \langle \mathtt{tree(y')}\rangle_4, \emptyset, \mathcal{I}_1\right) \\
i_2: & x_1 := x \to l; \\
 & \left(\mathtt{x_1} = \mathtt{x}' \wedge \mathtt{x} \neq \mathtt{nil} \,\vert\, \langle \mathtt{x} \mapsto [\mathtt{l}:\mathtt{x}', \mathtt{r}:\mathtt{y}']\rangle_2 * \langle \mathtt{tree(x')}\rangle_3 * \langle \mathtt{tree(y')}\rangle_4, \mathcal{F}_1 = i_2 \to \{2\}, \mathcal{I}_1\right) \\
i_3: & x_2 := x \to r; \\
 & \left(\mathtt{x_2} = \mathtt{y}' \wedge \mathtt{x_1} = \mathtt{x}' \wedge \mathtt{x} \neq \mathtt{nil} \,\vert\, \langle \mathtt{x} \mapsto [\mathtt{l}:\mathtt{x}', \mathtt{r}:\mathtt{y}']\rangle_2 * \langle \mathtt{tree(x')}\rangle_3 * \langle \mathtt{tree(y')}\rangle_4, \mathcal{F}_2 = \mathcal{F}_1[i_3 \to \{2\}], \mathcal{I}_1\right) \\
i_4: & x \to l := x_2; \\
 & \left(\mathtt{x_2} = \mathtt{y}' \wedge \mathtt{x_1} = \mathtt{x}' \wedge \mathtt{x} \neq \mathtt{nil} \,\vert\, \langle \mathtt{x} \mapsto [\mathtt{l}:\mathtt{x_2}, \mathtt{r}:\mathtt{y}']\rangle_2 * \langle \mathtt{tree(x')}\rangle_3 * \langle \mathtt{tree(y')}\rangle_4, \mathcal{F}_3 = \mathcal{F}_2[i_4 \to \{2\}], \mathcal{I}_1\right) \\
i_5: & x \to r := x_1; \\
 & \left(\mathtt{x_2} = \mathtt{y}' \wedge \mathtt{x_1} = \mathtt{x}' \wedge \mathtt{x} \neq \mathtt{nil} \,\vert\, \langle \mathtt{x} \mapsto [\mathtt{l}:\mathtt{x_2}, \mathtt{r}:\mathtt{x_1}]\rangle_2 * \langle \mathtt{tree(x')}\rangle_3 * \langle \mathtt{tree(y')}\rangle_4, \mathcal{F}_4 = \mathcal{F}_3[i_5 \to \{2\}], \mathcal{I}_1\right) \\
i_6: & \mathit{rotateTree}(x_1;); \\
 & \left(\mathtt{x_2} = \mathtt{y}' \wedge \mathtt{x_1} = \mathtt{x}' \wedge \mathtt{x} \neq \mathtt{nil} \,\vert\, \langle \mathtt{x} \mapsto [\mathtt{l}:\mathtt{x_2}, \mathtt{r}:\mathtt{x_1}]\rangle_2 * \langle \mathtt{tree(x_1)}\rangle_5 * \langle \mathtt{tree(y')}\rangle_4, \mathcal{F}_5 = \mathcal{F}_4[i_6 \to \{3,5\}], \mathcal{I}_2\right) \\
i_7: & \mathit{rotateTree}(x_2;); \\
 & \left(\mathtt{x_2} = \mathtt{y}' \wedge \mathtt{x_1} = \mathtt{x}' \wedge \mathtt{x} \neq \mathtt{nil} \,\vert\, \langle \mathtt{x} \mapsto [\mathtt{l}:\mathtt{x_2}, \mathtt{r}:\mathtt{x_1}]\rangle_2 * \langle \mathtt{tree(x_1)}\rangle_5 * \langle \mathtt{tree(x_2)}\rangle_6, \mathcal{F}_6 = \mathcal{F}_5[i_7 \to \{4,6\}], \mathcal{I}_3\right) \\
 & \} \\
\end{array}
$$

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 $\mathit{rotateTree}$

from these inherent limitations since it is based on detecting the *footprints* of statements. We illustrate this with the example in figure 4. In this case we have the procedure $\mathit{setBack}(x, y, z;)\,\{\mathit{local}\ x_1; c\}$, 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 pointer to be set for the head element. In this case we have the spec table with a single pre-condition $\langle \mathtt{ls}(x, y)\rangle_\bullet$ and single post-condition $\langle \mathtt{dls}(x, z, y, z')\rangle_\bullet$, 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

We have discussed how, in the case of the operational rule for specified commands, there is a need to infer the *frame assertion* in order to match the call-site assertion to the command's pre-condition. Given a call-site assertion $\Pi \,\vert\, \Sigma$ and command pre-condition $\Pi_1 \,\vert\, \Sigma_1$, the objective is to find a frame assertion Σ_F such that $\Pi \,\vert\, \Sigma \vdash \Pi_1 \,\vert\, \Sigma_1 * \Sigma_F$. We adapt the frame inference method of [2], which uses a proof theory for entailments between symbolic heaps. However, in our case, as well as inferring the formula, we also require that the frame assertion should correctly preserve its labels from the original call-site assertion since these are used to determine the footprint labels of the specified command. For this purpose we introduce the notion of *label respecting* entailment.

$$
\begin{array}{ll}
 & \left(\langle \mathtt{ls(x,y)}\rangle_1, \emptyset, \emptyset\right) \\
i_1: & \text{if}(x \neq y)\{ \\
 & \left(\mathtt{x}\neq\mathtt{y} \,|\, \langle \mathtt{ls(x,y)}\rangle_1, \emptyset, \emptyset\right) \\
 & \left(\mathtt{x}\neq\mathtt{y} \,|\, \langle \mathtt{x} \mapsto [\mathtt{n}:\mathtt{x}']\rangle_2 * \langle \mathtt{ls(x',y)}\rangle_3, \emptyset, \mathcal{I}_1\right) \\
i_2: & \quad x_1 := x \to n; \\
 & \left(\mathtt{x}_1=\mathtt{x}'\wedge\mathtt{x}\neq\mathtt{y} \,|\, \langle \mathtt{x} \mapsto [\mathtt{n}:\mathtt{x}']\rangle_2 * \langle \mathtt{ls(x',y)}\rangle_3, \mathcal{F}_1 = i_2 \to \{2\}, \mathcal{I}_1\right) \\
i_3: & \quad x \to b := z; \\
 & \left(\mathtt{x}_1=\mathtt{x}'\wedge\mathtt{x}\neq\mathtt{y} \,|\, \langle \mathtt{x} \mapsto [\mathtt{n}:\mathtt{x}', \mathtt{b}:\mathtt{z}]\rangle_2 * \langle \mathtt{ls(x',y)}\rangle_3, \mathcal{F}_2 = \mathcal{F}_1[i_3 \to \{2\}], \mathcal{I}_1\right) \\
i_4: & \quad setBack(x_1, y, x) \\
 & \left(\mathtt{x}_1=\mathtt{x}'\wedge\mathtt{x}\neq\mathtt{y} \,|\, \langle \mathtt{x} \mapsto [\mathtt{n}:\mathtt{x}', \mathtt{b}:\mathtt{z}]\rangle_2 * \langle \mathtt{dls(x_1,x,y,z')}\rangle_4, \mathcal{F}_3 = \mathcal{F}_2[i_4 \to \{3,4\}], \mathcal{I}_2\right) \\
 & \} \\
 & \text{where } \mathcal{I}_1 = \{\{2,1\},\{3,1\}\} \text{ and } \mathcal{I}_2 = \mathcal{I}_1 \cup \{\{4,3\},\{4,1\}\}
\end{array}
$$

Fig. 4. Independence detection for $setBack$

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

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

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

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 $E \mapsto [\rho]$, $\mathtt{ls}(E, F)$, $\mathtt{dls}(E, E_b, F, F_b)$ or $\mathtt{tree}(E)$. The guard $\mathtt{G}(\mathtt{op}(E))$ asserts that the heap is non-empty, and is defined as

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

$$\mathtt{G}(\mathtt{dls}(E, E_b, F_f, F_b)) \triangleq E \neq F_f \qquad \mathtt{G}(\mathtt{dls}(F_f, F_b, E_f, E)) \triangleq E \neq F_b$$

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 $\Pi \,|\, \Sigma$ and procedure pre-condition $\Pi_1 \,|\, \Sigma_1$. To find Σ_F such that $\Pi \,|\, \Sigma \vdash \Pi_1 \,|\, \Sigma_1 * \Sigma_F$, we apply the proof rules upwards starting from the entailment $\Pi \,|\, \Sigma \vdash \Pi_1 \,|\, \Sigma_1$, as instructed by the following theorem which we inherit from [2].

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

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

Then there is a complete proof of the label respecting entailment $\Pi \,|\, \Sigma \vdash \Pi_1 \,|\, \Sigma_1 * \Sigma_F$.

$$\frac{}{\Pi \,|\, \mathtt{emp} \vdash \mathtt{true} \,|\, \mathtt{emp}} \qquad \frac{\Pi \,|\, \Sigma \vdash \Pi' \,|\, \Sigma'}{\Pi \,|\, \Sigma \vdash \Pi' \wedge E = E \,|\, \Sigma'}$$

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

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

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

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

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

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

Fig. 5. Rules for label respecting entailment

When applying the label-respecting proof rules upwards, labels can only be removed from the left hand side of an entailment. Hence Σ_F will retain its labels from the call-site assertion $\Pi \,|\, \Sigma$. By theorem 1, the entailment $\Pi \,|\, \Sigma \vdash \Pi_1 \,|\, \Sigma_1 * \Sigma_F$ 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 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 command pre-condition is $\langle \mathtt{tree}(y) \rangle_\bullet$, the following 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 adapt an action trace semantics of programs from [6]. The action traces are composed of primitive actions α:

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

The $\mathtt{assume}(b)$ action is used to implement conditionals, as shown in the trace semantics of commands below. It filters out states which do not satisfy the boolean b. The $new_l(x)$ command allocates the location l if it is not already allocated. We choose this

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

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

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 $\{(s',h')\}$.

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

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

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

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

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

Fig. 7. Action trace semantics of commands

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 $(i,1) : \alpha$. For each specified command $i : \mathtt{com}(\mathcal{T})$, 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 $i : \mathtt{if}\ b\ c_1\ c_2$, we index the assume actions as $(i,1) : \mathtt{assume}(b)$ and $(i,1) : \mathtt{assume}(\neg b)$ 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 statements 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 $loc(\tau|_i, \tau, s, h) \cap loc(\tau|_{i'}, \tau, s, h) = \emptyset$.*

Given the trace model developed above, we can now formally state the soundness property of the independence detection algorithm given in figure 2.

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 complete proof of this result can be found in the technical report [14], and we give here an outline. 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*. A symbolic execution trace, $\mathcal{S}$, is a sequence of symbolic states related by applications of operational or rearrangement rules, beginning with some initial state ψ_I in the pre-condition and ending with some ψ_F in the final set of symbolic states that is used to test independences.

The concrete and symbolic executions are related by a notion of satisfaction between an action trace and a symbolic execution trace. An action trace τ satisfies a symbolic execution trace $\mathcal{S}$ if it is of the form $\tau|_{i_1}; ...; \tau|_{i_n}$, where $i_1, ..., i_n$ are the command indices for the operational rules that generate $\mathcal{S}$, and every intermediate concrete state in τ (after the execution of each index subtrace) satisfies the symbolic heap in the corresponding symbolic states. Thus this notion of satisfaction depends only on the symbolic heap component of symbolic states and not on the footprint and intersection logs. By soundness of standard symbolic execution [2], we have that every concrete trace of the program satisfies *some* symbolic execution trace generated by the algorithm.

This relation connecting concrete and symbolic executions is then used to interpret the labels in the symbolic heaps and the footprint and intersection logs. The underlying idea is that, given a concrete initial state (s, h) and an action trace τ satisfying a symbolic execution trace $\mathcal{S}$, every label l occurring in any of the symbolic states in $\mathcal{S}$ corresponds to a *fixed* set of heap locations throughout the entire concrete execution of τ from (s, h). This location set, denoted $labloc(l, \mathcal{S}, \tau, s, h)$, is used to reason about the heap locations that the labels in the footprint and intersection logs represent. We show that for any two subtraces $\tau|_i$ and $\tau|_{i'}$ of τ, if the footprint labels of i and i' in the final footprint log of $\mathcal{S}$ do not intersect according to the final intersection log of $\mathcal{S}$, then the two subtraces access disjoint locations, that is, $loc(\tau|_i, \tau, s, h) \cap loc(\tau|_{i'}, \tau, s, h) = \emptyset$.

The algorithm determines two commands with indices i and i' to be independent if they have non-intersecting footprint labels according to each of the final symbolic states generated by the algorithm. Since every action trace satisfies some symbolic execution trace, we have that in every action trace of the program, the subtraces of i and i' access disjoint locations, which means that i and i' are independent by definition 6.

8 Conclusion and Future Work

In this work we have focussed on laying the foundations of our extended separation logic framework for independence detection. We plan to extend the method we describe to the more complex data structures handled by separation logic shape analyses [1], to integrate our method with the existing *space invader* tool for shape analysis [17,4], 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. Following this, we intend to investigate the application of our method to other kinds of program optimizations.

Acknowledgements. We thank the anonymous referees for very helpful comments. Raza acknowledges support of an ORS award and EPSRC grant "Smallfoot: static assertion checking for C programs". Gardner acknowledges support of a Microsoft Research Cambridge/Royal Academy of Engineering Senior Research Fellowship. Calcagno acknowledges support of an EPSRC advanced fellowship.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., OHearn, P., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic Execution with Separation Logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Automatic modular assertion checking with separation logic. In: 4th FMCO (2006)
4. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional Shape Analysis. In: POPL (2009)
5. Distefano, D., O'Hearn, P., Yang, H.: A Local Shape Analysis based on Separation Logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
6. Calcagno, C., O'Hearn, P., Yang, H.: Local Action and Abstract Separation Logic. In: LICS (2007)
7. Ghiya, R., Hendren, L.J., Zhu, Y.: Detecting Parallelism in C programs with recursive data structures. In: Koskimies, K. (ed.) CC 1998. LNCS, vol. 1383. Springer, Heidelberg (1998)
8. Gupta, R., Pande, S., Psarris, K., Sarkar, V.: Compilation Techniques for Parallel Systems. In: Parallel Computing (1999)
9. Hendren, L.J., Nicolau, A.: Parallelizing programs with recursive data structures. In: IEEE Transactions on Parallel and Distributed Systems (1990)
10. Hummel, J., Hendren, L.J., Nicolau, A.: A general data dependence test for dynamic, pointer-based data structures. In: PLDI (1994)
11. Hoare, T., O'Hearn, P.: Separation Logic Semantics of Communicating Processes. In: FICS (2008)
12. Horwitz, S., Pfeiffer, P., Reps, T.W.: Dependence analysis for poiner variables. In: PLDI (1989)
13. Marron, M., Stefanovic, D., Kapur, D., Hermenegildo, M.: Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models. In: LCPC (2008)
14. Raza, M., Calcagno, C., Gardner, P.: Automatic Parallelization with Separation Logic. Imperial College Technical Report DTR08-16 (2008)
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th LICS (2002)
16. Rinard, M.C., Diniz, P.C.: Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. In: ACM Transactions on Programming Languages and Systems (1997)
17. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable Shape Analysis for Systems Code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)