Making fast hardware with separation logic

Byron Cook
Microsoft Research

Stephen Magill
Carnegie Mellon University

Mohammad Raza
Imperial College

Jiri Simsa
Carnegie Mellon University

Satnam Singh
Microsoft Research

Abstract

Recently developed tools now allow us to automatically synthesize hardware from programs that use the heap [5]. Unfortunately, because current tools are unable to accurately track the heap-carried data dependencies between program commands, the synthesis tools miss many opportunities for optimization, such as pipelining, parallelization, or memory localization. Thus, the resulting circuits have poor performance. In this paper we describe a separation logic based program analysis for identifying heap-carried data dependencies between program statements, and demonstrate the performance gains that it enables when performing hardware synthesis.

1. Introduction

Recent work on high-level synthesis tools now make it possible to take programs making non-trivial use of dynamically-allocated heap (*e.g.* linked-list C programs that call `malloc` and `free`) and convert them to hardware circuits [5]. Unfortunately, the resulting circuits are slow. The problem is that the program's heap makes it more difficult to work out when circuit-level pipelining or parallelization can be applied without changing the behavior of the program. Furthermore, without a better understanding of each command's heap footprint [23], we are forced to build connections between each circuit implementing a command and every possibily allocated heap cell in the circuitry used to represent the heap.

What's needed is a tool that automatically works out layout of the heap during the program's execution and then provides accurate information about heap-carried dependecies between commands (thus allowing pipelining and parallelisation) as well as symbolic memory footprints [23] (thus allowing the heap to be distributed across multiple memories on chip).

In this paper we develop a new program analysis that infers precisely this information. Using an abstract domain inspired by a recent extension to separation logic [24], our analysis automatically infers program invariants that describe the layout of linked data structures during the program's execution, (*e.g.* doubly-linked lists, or trees), as well as information about which program commands depend on each other via connections in the heap. These invariants symbolically track where memory cells are allocated, deallocated, read from, and written to.

Using the output of our analysis we can then more accurately determine when it is safe to employ circuit transformations designed to optimize performance. Note that code fragments must respect conditions regarding dangling pointers and termination before we can soundly apply the optimizations. We make these conditions clear when proving soundness.

Related work. Recent work [24] has developed a program logic designed to prove properties about dependences in heap-manipulating programs. In this work we develop a program analysis using an abstract domain influenced by [24]. Note that, beyond developing an analysis, in order to express information about pipelining we extended the logic to determine loop carried dependences. Our approach also uses a new notion of heap labelling which avoids the bookkeeping overhead of footprint and intersection logs of [24].

Previously reported program analysis (*e.g.* pointer analysis) can be used to soundly overapproximate the information infered here. In fact, [5] does precisely this. Experimentally we have found these analyses to be too imprecise, limiting the performance of the resulting circuits. Note that these previously reported analyses are more scalable than the one we propose here. We trade precision for analysis performance, and thus can only synthesize programs of relatively modest size.

Numerous heap dependence analyses have been developed (*e.g.* [10, 13, 14, 19]), and we present a similar analysis based on separation logic, which provides precise aliasing information about heap data structures to give accurate dependence information. Dependence analysis is a standard analysis which is useful in different applications such as optimization, program slicing and program integration. However, we describe here how a standard heap dependence analysis needs to be strengthened before it can be used to soundly optimize programs. Program optimizations have been implemented based on heap dependence analyses [10, 15], using the *independence assumption*, which states when two commands are independent (access separate heap and variables in all possible executions), then they can be parallelized or reordered to give an equivalent program. We show in Section 4 that the independence assumption is unsound in general, and present restrictions on the analysis which guarantee correctness of the optimizations. In [15] specific rewrite rules are used convert sequential code into parallel code. The applicability of these rewrites depend in part on the structure of the input code, while our approach uses traditional compiler

```
1      while (1) {
2          k = n;
3          l1 = NULL;
4          l2 = NULL;
5
6          while (k>0) {
7              l1 = insertion_sort(l1,input(i1));
8              l2 = insertion_sort(l2,input(i2));
9              k−−;
10         }
11
12         while (l1 != NULL && l2 != NULL) {
13             output(o,remove_smallest(l1,l2));
14         }
15     }
```

Figure 1. C language description of a circuit that reads in n inputs from input streams i1 and i2 and outputs them in sorted order to output stream o.

techniques to discover maximal parallelism given a set of control and data dependencies of the input program.

Our soundness results in section 4 are an investigation of independence in the presence of dynamic memory allocation and deallocation. This has some relation to work on independence models [25], especially [12], where examples based on allocation and deallocation are presented to show interaction between parallel processes and as counterexamples to the completeness of concurrent separation logic. In contrast, we have investigated the sense of equivalence which can be guaranteed under such interactions and the constraints required for this equivalence to hold. In relation to [17], we are showing that certain allocations and deallocations are not movers, and the conditions under which they can be 'moved' are based on an extension of the heap footprint of commands.

Previous work based on rewriting has been used succesfully to optimize (as well as prove the correctness) of pipelined and superscalar circuits (*e.g.* [20]). The difficulty in these algebraic rewrite systems is that, in many cases, the rules cannot be used unless the lack of aliasing can be established. Our work tackles precisely this problem. As future work it is worth investigating an integration of our analysis with rewrite based circuit optimization techniques.

2. Example

Consider the program in Figure 1, which provides a C-level description of a circuit designed to read n values from the input signals i1 and i2 and write them out in sorted order on the output signal o. We assume that insertion_sort is the standard list-based procedure for insertion sort. The procedure remove_smallest is defined as:

```
int remove_smallest(LIST ∗ p, LIST ∗ q) {
    if (q==NULL ||
        (p != NULL && (p−>value < q−>value))) {
        int x = p−>value;
        LIST ∗ t = p;
        p=p−>next;
        free(p);
        return x;
    } else {
        int x = q−>value;
        LIST ∗ t = q;
        q=q−>next;
        free(q);
        return x;
    }
}
```

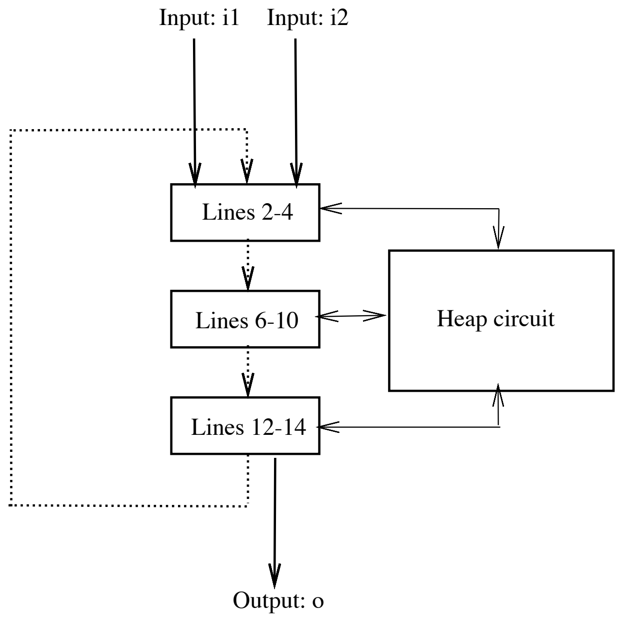

Figure 2. Top-level schematic of unoptimized circuit from program in Figure 1. Dotted lines indicate enable wires. Solid lines indicate data wires and buses.

Recent work [5] allows us to synthesize this heap-based program to hardware. The trick is to figure out that, in this case, the code never needs more than 2n linked-list nodes at any time during execution, thus leading to a finite-state system (for a fixed n).

The problem is that the resulting hardware is not effecient: Without a deep understanding of the invariants about the heap's structure, for the sake of correctness, we are forced to build a circuit whose layout effectively matches the control-flow-graph of the original program. In this conservative model, only one command sub-circuit is allowed to execute at any given time.

See Figure 2 for a high-level schmetic view of the circuit produced using techniques from [5]. The boxes with annotations such as "Lines 6-10" represent sub-circuits implementing the corresponding commands and loops. Each of these sub-circuits begins executing when its enable input is set to true, and sets its neighbour's enable wire to true when it is done executing. The fact that only one command sub-circuit executes at a time leads to poor latency. The circuit's throughput is also poor, as without an understanding of the heap-based feedback between loop iterations we cannot safely put pipeline delays in between sub-circuits representing lines 6-10 and lines 12-14. Without information about which heap cells a command might read from or write to, we must also build a connection between every command and the circuit that represents the program's heap. The fact that the memory must be wired up to each sub-circuit can lead to long wires, which may inhibit clock speeds.

Now consider Figure 3, which contains a high-level schematic view of an optimized circuit resulting from our method. Because the heap-footprint of the commands at lines 7 and 8 are independent and the order in which values are read from $i1$ and $i2$ is irrelevant, we can break the loop at lines 6-10 into two loops: L1 consisting of lines 2, 3, 6, 7, 9, and 10 and L2 consisting of lines 2, 4, 6, 8, 9, and 10. These two loops are independent (after α-renaming of the variable k), which allows us to execute them in parallel. Note that because the new loop L1 only touches the heap segment representing the list pointed to by l1, we can use a local memory circuit to store this list, rather than a shared representation of the entire program heap. Analagously we can do the same for the list pointed to by l2. When both of the loops have completed sorting values from the input signals, the enable line should be activated, thus we have wired the enable line from the pipeline stage back to the circuits implementing L1 and L2.

Our analysis also tells us that once we have built up the lists pointed to by l1 or l2 we are free to make a copy of them for the use of circuit implementing lines 12-14 and immediately begin rebuilding the lists l1 or l2 for the next iteration of the top-level loop. This is achieved by inserting a double-buffered pipeline stage.

Our experimental evalution has shown that using heap dependencies computed by our analysis to parallelize a sequential circuit that implements the above program has decreased latency by 65% and increased throughput by 183%. Further pipelining and memory localization of the circuit enabled by our analysis has improved throughput by 283% with respect to its sequential version.

3. Analysis

In this section we describe our separation logic based analysis for identifying heap-carried data dependencies between program statements. We illustrate our analysis in the setting of a simple while language and predicates for linked lists, as is standard in the literature on program analysis with separation logic [23].

3.1 Labelled Symbolic Heaps

We describe our approach in the simple storage model of [8], which is based on a set Loc of locations and $\texttt{Val} = \texttt{Loc} \cup \{0\}$. We assume a finite set Var of program variables and an infinite set Var′ of primed variables.

$x, y, ..$	$\in$	$\texttt{Var}$	program variables
$x', y', ..$	$\in$	$\texttt{Var}'$	primed variables

Primed variables will not be used in programs, only within formulae where they will be implicitly existentially quantified. We set

$$\begin{aligned}\texttt{Heaps} &= \texttt{Loc} \rightharpoonup_{fin} \texttt{Val}\\ \texttt{Stacks} &= (\texttt{Var} \cup \texttt{Var}') \to \texttt{Val}\\ \texttt{States} &= \texttt{Stacks} \times \texttt{Heaps}\end{aligned}$$

The analysis shall use the standard fragment of separation logic known as *symbolic heaps*OB [1, 8].

E, F	$::=$	$0 \mid x \mid x'$	expressions
Π	$::=$	$\texttt{true} \mid E = E \mid E \neq E \mid \Pi \wedge \Pi$	pure formulae
S	$::=$	$E \mapsto F \mid \texttt{ls}(E, F)$	simple spatial formulae
Σ	$::=$	$\texttt{emp} \mid S \mid \Sigma * \Sigma$	spatial formulae
SH	$::=$	$\Pi \mid \Sigma$	symbolic heaps

The interpretation, $s, h \models \Pi \mid \Sigma$, is the standard one on stack s and heap h. Expressions are program or logical variables or 0. Pure formulae are a conjunction of equalities or inequalities of expressions interpreted on s, while spatial formulae specify properties of h. The predicate emp holds in the empty heap where nothing is allocated. The formula $\Sigma_1 * \Sigma_2$ uses the separating conjunction of separation logic. It holds in h if h can be split into two disjoint parts $h1$ and $h2$ such that Σ_1 holds in $h1$ and Σ_2 in $h2$. The points-to assertion $E \mapsto F$ describes a heap with single allocated address with contents F. The formula $\texttt{ls}(E, F)$ holds for a linked list segment from E to F. All primed variables in symbolic heaps are existentially quantified.

Our analysis will be required to track portions of the heap during execution in order to determine dependencies between program statements. We shall do this with a notion of *labeling* to keep track of portions of the heap, and associate every simple spatial formula in a symbolic heap with a set of labels from a fixed a set of Lab.

l	$\in$	$\texttt{Lab}$	labels
L	$\in$	$P(\texttt{Lab})$	label sets
Λ	$::=$	$\texttt{emp} \mid \langle S \rangle_L \mid \Lambda * \Lambda$	labelled spatial formulae
LSH	$::=$	$\Pi \mid \Lambda$	labelled symbolic heaps

We write $unlabelled(\Lambda)$ to represent the unlabelled formula that is Λ without the label sets. A label l appearing in a formula represents a set of allocated heap locations. Labels can, for example, be indices of program statements, representing the part of the heap that the statement accessed in the symbolic execution. For example, a formula $\langle \texttt{ls}(x, 0) \rangle_{\{19,42\}} * \langle \texttt{ls}(y, 0) \rangle_{\{3\}}$ at some point in the symbolic execution may mean that statements 42 and 19 accessed list x but not list y, while statement 3 accessed list y but not list x. Labels need not necessarily be statement indices, and can in general be used as arbitrary 'markers' on the heap.

For a label l and formula Λ, we write $\Lambda|_l$ to be the conjunction of all formulae in Λ whose label sets contain l. We can have a simple formal interpretation of labels by extending heaps to contain a label set for each heap cell in the heap. This will collect the labels of all the commands that access the heap cell. We extend heaps with label sets as follows:

$$\texttt{Heaps} = \texttt{Loc} \rightharpoonup_{fin} (\texttt{Val} \times P(\texttt{Lab}))$$

So every heap cell has a value and a label set. The satisfaction of unlabelled symbolic heaps, $s, h \models \Pi \mid \Sigma$, just ignores the label sets in h. For labelled symbolic heaps, we have that the labels of every heap cell are contained in the label set of the part of the symbolic heap that describes the cell.

DEFINITION 1 (Label Satisfaction). *We have $s, h \models \Pi \mid \langle S \rangle_L$ iff $s, h \models \Pi \mid S$ and for all $l \in dom(h)$, if $h(l) = (l', L')$ then $L' \subseteq L$. We have that $s, h \models \Pi \mid \Lambda_1 * \Lambda_2$ iff $h = h1 * h2$ such that $s, h_1 \models \Pi \mid \Lambda_1$ and $s, h_2 \models \Pi \mid \Lambda_2$.*

Notice that a symbolic heap only gives an over-approximation of the labels of a concrete heap. For example, assume we have s, h where h contains separate lists at x and y, the head of the list at x has label set $\{l\}$, and the label sets of the rest of the heap are all empty. Then we have

$$\begin{aligned}s, h &\models \langle x \mapsto x' \rangle_{\{l\}} * \langle \texttt{ls}(x', y) \rangle_\emptyset * \langle \texttt{ls}(y, 0) \rangle_\emptyset\\ s, h &\models \langle \texttt{ls}(x, y) \rangle_{\{l\}} * \langle \texttt{ls}(y, 0) \rangle_{\{l\}}\\ s, h &\not\models \langle \texttt{ls}(x, y) \rangle_\emptyset * \langle \texttt{ls}(y, 0) \rangle_\emptyset\end{aligned}$$

Such over-approximation may, for example, be encountered in the analysis when we fold predicates.

3.2 Analysis

We consider a simple imperative programming language with heap manipulating commands and while loops.

$$\begin{aligned}b &::= E = E \mid E \neq E\\ A &::= x := E \mid x := [E] \mid [E_1] := E_2 \mid \texttt{new}(x) \mid \texttt{dispose}(x)\\ c &::= A \mid \texttt{if}\ b\ c_1\ c_2 \mid \texttt{while}\ b\ c \mid c_1; c_2\end{aligned}$$

We assume that every command in the program is labelled with a unique index, and we shall use label variables $l, l_1, l_2, ...$ to range over these index labels. A program is then a sequential composition of the form

$$l_1 : c_1; \ldots ; l_n : c_n$$

where every c_i is an atomic command, or a composite command (conditional or while loop), and l_i denote the index labels of the commands. The nested bodies of composite commands are also sequential blocks of this form. We first describe the dependency analysis between commands in a sequential block, and will later describe the detection of loop-carried dependencies (dependencies between iterations of loops), which is needed for pipelining.

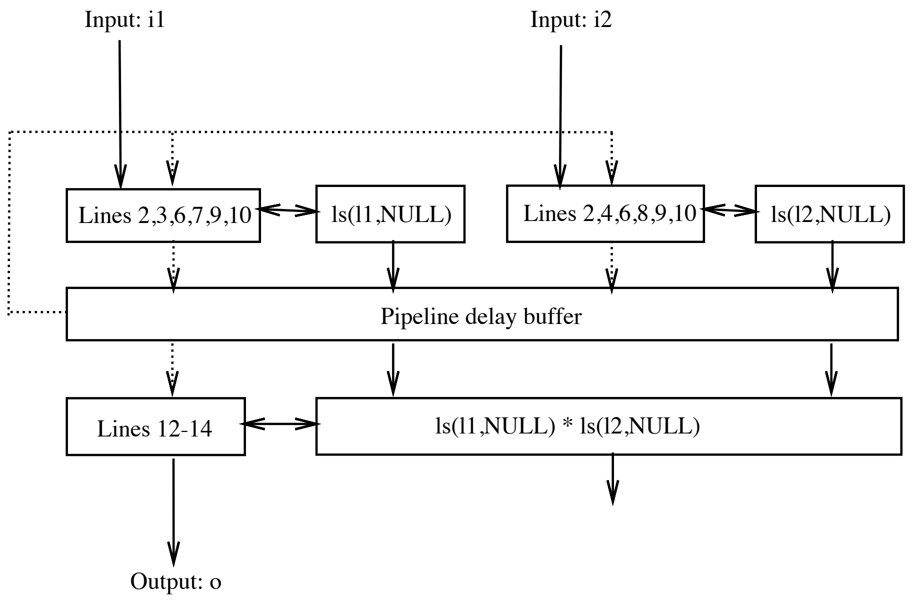

Figure 3. Top-level schematic of optimized circuit from program in Figure 1. Dotted lines indicate enable wires. Solid lines indicate data wires and buses.

For every sequential block in the program (at any level of nesting inside loops or conditionals), our goal is to detect when any two component commands $l_i : c_i$ and $l_j : c_j$ access the same heap locations. This dependency information can be represented as a relation on the command labels. We describe our method for a single notion of heap *access*, and shall later describe how it is a simple extension to differentiate between read and write access.

In general, our analysis can be based on any standard shape analysis with separation logic [1], and in this work we have implemented it on top of the analysis from [18]. The basic idea is to extend the shape analysis to track labels through the symbolic execution and to detect the heap footprints of commands in terms of labels. The propagation of labels in the symbolic execution is illustrated in Figure 4.

The dependence detection algorithm is given in Figure 5. Given a sequential block $c = l_1 : c_1; \ldots; l_n : c_n$ and a set of labelled symbolic heaps pre, it detects dependencies between the component commands of c by tracking their labels through a symbolic execution of c starting from the given pre-condition states. For every label l in the sequential block, the algorithm determines a set of dependencies $deps(l)$ of all labels on which l depends. We describe the algorithm assuming sequential blocks with atomic commands, and address composite commands in the next section.

The $rearrange(pre, E)$ procedure makes the cell at E explicit in the states in pre using the rearrangement rules in Figure 4. This is necessary for an atomic command accessing cell E to execute. For example, if the command $y := [E]$ then we may have

$$\begin{aligned} & rearrange(\{x = E \wedge x \neq 0 ¦ \langle \mathtt{ls}(x, 0)\rangle_l\}, E) \\ = {} & \{x = E \wedge x \neq 0 ¦ \langle E \mapsto x'\rangle_l * \langle \mathtt{ls}(x', 0)\rangle_l\} \end{aligned}$$

The $getFootprintLabels(c', \Pi ¦ \Lambda)$ procedure detects the part of $\Pi ¦ \Lambda$ that c' is accessing, and returns the labels associated with that part. For atomic commands that do not access existing heap cells, such as assignment or allocation, this set is empty. For atomic commands accessing cell E, the state is in the rearranged form $\Pi ¦ \langle E \mapsto F\rangle_L * \Lambda$, so the computed footprint labels are L. The

EXECUTION RULES

$$\frac{\Pi ¦ \Lambda}{x = E[x'/x] \wedge (\Pi ¦ \Lambda)[x'/x]} \; l : x := E, x' \mathit{fresh}$$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto F'\rangle_L}{\Pi ¦ \Lambda * \langle E \mapsto F\rangle_{L \cup \{l\}}} \; l : [E] := F$$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto F\rangle_L}{x = F[x'/x] \wedge (\Pi ¦ \Lambda * \langle E \mapsto F\rangle_{L \cup \{l\}})[x'/x]} \; \dagger$$

$$\dagger \; l : x := [E], x' \mathit{fresh}$$

$$\frac{\Pi ¦ \Lambda}{(\Pi ¦ \Lambda)[x'/x] * \langle x \mapsto 0\rangle_{\{l\}}} \; l : \mathtt{new}(x), x' \mathit{fresh}$$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto F\rangle_L}{\Pi ¦ \Lambda} \; l : \mathtt{dispose}(E)$$

REARRANGEMENT AND FOLDING RULES

$$\frac{\Pi ¦ \Lambda * \langle F \mapsto F'\rangle_L}{\Pi ¦ \Lambda * \langle E \mapsto F'\rangle_L} \; \Pi \vdash E = F$$

$$\frac{\Pi ¦ \Lambda * \langle \mathtt{ls}(F, F')\rangle_L}{\Pi ¦ \Lambda * \langle E \mapsto x'\rangle_L * \langle \mathtt{ls}(x', F')\rangle_L} \; \dagger$$

$$\dagger \; \Pi ¦ \Lambda * \langle \mathtt{ls}(F, F')\rangle_L \vdash F \neq F' \wedge E = F \text{ and } x' \text{ fresh}$$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto x'\rangle_{L_1} * \langle \mathtt{ls}(x', F)\rangle_{L_2}}{\Pi ¦ \Lambda * \langle \mathtt{ls}(E, F)\rangle_{L_1 \cup L_2}}$$

Figure 4. Label propagation in symbolic execution

```
$getDeps(c, pre) =$
  **if** $c$ is empty then **return**
  **else** let $c = l : c'; c''$
    **if** $c'$ is an atomic command accessing heap cell $E$
    **then** $pre = rearrange(pre, E)$
    **for all** $\Pi ¦ \Lambda \in pre$
      $deps(l) = getFootprintLabels(c', \Pi ¦ \Lambda)$
    **end for**
    $post = getPost(c', l, pre)$
    $getDeps(c'', post)$
```

Figure 5. Dependency detection algorithm

$$
\begin{array}{l}
\{x \neq 0 ¦ \langle \mathtt{ls}(x,0)\rangle_{\emptyset}\} \\
\quad l_1 : \mathtt{new}(y); \\
\{x \neq 0 ¦ \langle \mathtt{ls}(x,0)\rangle_{\emptyset} * \langle y \mapsto 0\rangle_{\{l_1\}}\} \\
\{x \neq 0 ¦ \boxed{\langle x \mapsto x'\rangle_{\emptyset}} * \langle \mathtt{ls}(x',0)\rangle_{\emptyset} * \langle y \mapsto 0\rangle_{\{l_1\}}\} \\
\quad l_2 : z := [x]; \\
\{z = x' \wedge x \neq 0 ¦ \boxed{\langle x \mapsto x'\rangle_{\{l_2\}}} * \langle \mathtt{ls}(x',0)\rangle_{\emptyset} * \langle y \mapsto 0\rangle_{\{l_1\}}\} \\
\quad l_3 : \mathtt{dispose}(x); \\
\{z = x' \wedge x \neq 0 ¦ \langle \mathtt{ls}(x',0)\rangle_{\emptyset} * \boxed{\langle y \mapsto 0\rangle_{\{l_1\}}}\} \\
\quad l_4 : [y] := z; \\
\{z = x' \wedge x \neq 0 ¦ \langle \mathtt{ls}(x',0)\rangle_{\emptyset} * \langle y \mapsto z\rangle_{\{l_1, l_4\}}\}
\end{array}
$$

Figure 6. Dependency detection in replacing the head of a list

case when c' is a composite command, like a conditional or a loop, is described in the next section. We add the footprint labels to the set of dependencies $deps(l)$ of l.

The next step is to propagate the labels by calling $getPost(c', l, pre)$, which returns the set of symbolic post states after executing c' on pre, with the label l added to the part of the post-states that c' accessed. For atomic commands this is specified by the execution rules in Figure 4. For composite commands we use frame inference to do this propagation. The details of how frame inference is used to carry out the propagation are given in the next section. Finally, the main procedure is recursively called on the remaining block starting from the set of post states.

We illustrate the method in Figure 6, which shows the label propagation for a sequence of commands to replace the head of linked list. The footprints of commands are boxed in the command's pre-state. l_2 does not depend on l_1 because its footprint labels are the label set of the boxed formula $(x \mapsto x')$ in the pre-state of l_2, which is empty. l_3 depends on l_2 but not on l_1, and l_4 depends only on l_1. Note that there is a stack dependency between l_4 and l_2 because of variable z, but our goal here is only to determine heap dependencies. Stack dependencies can be easily determined from the syntax of commands.

To distinguish between read and write access, we can annotate labels with the type of access. Thus for a command with label l we can have labels l_r and l_w to represent and read and write access, and insert l_r into parts of the symbolic heap that the command only reads. This way we can prevent a dependency between two commands that only read the same location.

The dependency detection algorithm can be applied to the main program, and also independently to the bodies of all composite commands such as loops and conditionals. However, for a given sequential block with composite commands, we have yet to describe how the algorithm detects the footprint of the whole command in the pre-state, and also how the labels are propagated to the post-states of the whole command. This is done through a process of inferring the *frame* of the entire command.

3.3 Frame inference

We now describe the footprint detection (*getFootprintLabels*) and label propagation (*getPost*) for composite commands. Given a composite command $l : c$ and a pre-state $\Pi ¦ \Lambda$, we need to find the part of Λ accessed by c in its entire execution, and also the set of post-states in which the labels in the pre-state and the label l of c are propagated correctly. This is achieved by a process of frame inference on the symbolic execution of c that is provided by the shape analysis. The frame inference method does not require label tracking, as the aim is only to search for a part of the initial state that is never accessed in the execution of the command. So for the unlabelled pre-state $\Pi ¦ \Sigma$ (where $\Sigma = unlabelled(\Lambda)$), the frame inference finds a Σ_F such that $\Sigma = \Sigma' * \Sigma_F$, and Σ_F is never accessed in the symbolic execution of c.

For example, say we are given the pre-state $t = x ¦ \mathtt{ls}(x, y) * \mathtt{ls}(y, 0)$ and the loop $\mathtt{while}\ (t \neq y)\ \{t := [t]\}$. After symbolic execution and folding on the first couple of unrollings of the loop, we obtain the state

$$\mathtt{ls}(x,t) * \mathtt{ls}(t,y) * \mathtt{ls}(y,0)$$

So far we have determined that $\mathtt{ls}(y, 0)$ has not been accessed. Then in the invariant execution we have

$$
\begin{array}{l}
\{t \neq y ¦ \mathtt{ls}(x,t) * \mathtt{ls}(t,y) * \mathtt{ls}(y,0)\} \\
\{\mathtt{ls}(x,t) * t \mapsto t_1' * \mathtt{ls}(t_1',y) * \mathtt{ls}(y,0)\} \\
\quad t := [t]; \\
\{\mathtt{ls}(x,t_2') * t_2' \mapsto t * \mathtt{ls}(t,y) * \mathtt{ls}(y,0)\} \\
\{\mathtt{ls}(x,t) * \mathtt{ls}(t,y) * \mathtt{ls}(y,0)\}
\end{array}
$$

Thus $ls(y, 0)$ is not unfolded or accessed in the invariant execution, so no iteration of the loop accesses the concrete heap that satisfies $ls(y, 0)$. However, we also need to ensure that if the invariant is reestablished at the end of the execution, then none of the accessed heap moves over to the frame assertion. We ensure this by not allowing the frame assertion to contain existential variables, so that all the predicates in the frame assertion are *precise* [22]. A precise predicate is such that for any concrete heap, there is at most one subheap that satisfies the predicate.

Formally, assume we have the invariant $\Pi ¦ \Sigma * \Sigma_F$, where Σ_F is precise, and the concrete heap is $h * h_F$ at the beginning of an iteration such that $h \models \Sigma$ and $h_F \models \Sigma_F$. Then after the symbolic execution we have $\Pi' ¦ \Sigma' * \Sigma_F$ and the concrete state $h' * h_F$ where $h' \models \Sigma'$ and $h_F \models \Sigma_F$, and h_F has not been accessed. To get back the invariant, we use the entailment:

$$\Pi' ¦ \Sigma' * \Sigma_F \vdash \Pi ¦ \Sigma * \Sigma_F$$

The fact that Σ_F is precise ensures that only h_F satisfies Σ_F in the reestablished invariant. So, in every iteration, h_F is the only concrete heap that satisfies Σ_F and it is not accessed in any iteration of the loop.

Once we have determined the frame assertion we use it to obtain the footprint labels from the pre-state, and also to propagate the labels from the pre-state of the whole command to its post-states, as follows. Assume that the labelled pre-state is $\Pi ¦ \Lambda * \Lambda_F$ and the unlabelled pre-state is $\Pi ¦ \Sigma * \Sigma_F$. The set of footprint labels of the command is obtained as union of the labels in Λ, that is, all labels not included in the frame assertion.

For the unlabelled pre-state $\Pi ¦ \Sigma * \Sigma_F$, every post-state from the shape analysis is of the form $\Pi' ¦ \Sigma' * \Sigma_F$. In this post-state we label Σ_F exactly as Λ_F, and for all the formulae in Σ', we assign them the label set $Ft \cup \{l\}$, where Ft are the footprint labels and l is the label of the composite command. The following example illustrates our frame inference.

$\{t = x \mid \langle \mathtt{ls}(x,y)\rangle_{L_1} * \langle \mathtt{ls}(y,0)\rangle_{L_2}\}$
$l_1 : \mathtt{while}\ (t \neq y)\ \{t := [t]\}$
$\{t = y \mid \langle \mathtt{ls}(x,y)\rangle_{L_1 \cup \{l_1\}} * \langle \mathtt{ls}(y,0)\rangle_{L_2}\}$
$l_2 : \mathtt{while}\ (t \neq 0)\ \{t := [t]\}$
$\{t = 0 \mid \langle \mathtt{ls}(x,y)\rangle_{L_1 \cup \{l_1\}} * \langle \mathtt{ls}(y,0)\rangle_{L_2 \cup \{l_2\}}\}$

In this case the frame assertion inferred for l_1 is $\mathtt{ls}(y, 0)$ and for l_2 it is $\mathtt{ls}(x, y)$, and so the footprint set of l_1 is L_1 and for l_2 it is L_2.

3.4 Loop carried dependencies

So far we have described the dependence analysis for commands in a sequential block. This could be applied to the bodies of loops, but it only finds dependences *within* any iteration of the loop. To do optimizations such as pipelining, we need to determine heap dependences *between* iterations of the loop. This can be determined by performing a fixpoint computation on the labelled symbolic heaps to reach an invariant which distinguishes parts of the symbolic heap that have not been accessed so far in the loop, that is, have an empty label set.

Assume that we are given a loop $\mathtt{while}(B)\{C\}$ and an initial set of states with empty label sets, and our goal is to determine whether there is a heap dependency between any two iterations of the loop. In order to find dependencies between iterations, we assume distinct labels to the instances of commands in each iteration of the body C, so that iteration i is of the form

$$l_{i,1} : C_1; \ldots ; l_{i,n} : C_n$$

To determine an invariant, we perform a label propagating execution on the loop starting with the first iteration, and at the widening stage at the end of each iteration we replace all non-empty label sets with $\top$. The $\top$ element is the set of all labels, so the abstraction only maintains information about the parts of the state that "have been accessed by some command", as opposed to not accessed by any command. We stop when we reach a state Λ such that $\Lambda \vdash \Lambda'$ for some Λ' at the end of a previous iteration. So the invariant that is reached will have label sets that are either empty or $\top$. For example, for the loop $\mathtt{while}\ (t \neq 0)\ \{t := [t]\}$ with initial state $\{t = x \mid \langle \mathtt{ls}(x, 0)\rangle_\emptyset\}$, we will reach the invariant

$$\{\langle \mathtt{ls}(x,t)\rangle_\top * \langle \mathtt{ls}(t,0)\rangle_\emptyset\}$$

When we have determined the invariant, we can then execute the body of an arbitrary iteration on it to determine whether any command in the iteration accesses a state with label set $\top$. So for our example we have the following execution:

$\{t \neq 0 \mid \langle \mathtt{ls}(x,t)\rangle_\top * \langle \mathtt{ls}(t,0)\rangle_\emptyset\}$
$\{\mid \langle \mathtt{ls}(x,t)\rangle_\top * \boxed{\langle t \mapsto t'\rangle_\emptyset} * \langle \mathtt{ls}(t',0)\rangle_\emptyset\}$
$l_{i,1} :\ t := [t];$
$\{\mid \langle \mathtt{ls}(x,t'')\rangle_\top * \langle t'' \mapsto t\rangle_{\{l_{i,1}\}} * \langle \mathtt{ls}(t,0)\rangle_\emptyset\}$
$\{\langle \mathtt{ls}(x,t)\rangle_\top * \langle \mathtt{ls}(t,0)\rangle_\emptyset\}$

The footprint label set of command instance $l_{i,1}$ is not $\top$, which means that $l_{i,1}$ accesses a part of the heap that has not been accessed by any command instances from previous iterations. Thus we determine that there is no heap dependency between iterations of the loop, although note that there is a stack dependency due to variable t. In the case of the merge program in Figure 1, for the infinite outer loop the heap is empty before and after every iteration, which is a simple case of heap independence between iterations.

4. Restrictions and soundness

In the previous section we have described our method for heap dependency analysis. Note that other approaches[10, 13, 14, 19, 24] have been used in the past to carry out the same type of analysis. Program optimizations have been proposed using this type of analysis [10, 15], based on the *independence assumption*, which states that if two commands are independent (access separate heap and variables in all possible executions), then they can be parallelized or reordered to give an equivalent program. We describe here how the independence assumption is, in general, not a sound basis for such optimizations: transformations based on this assumption can produce results that are not equivalent to the original program. We then propose a restriction on input programs as well as the application of the transformations, and demonstrate the soundness of our restricted method using a trace semantics of programs.

4.1 Dynamic memory allocation

Dynamic memory allocation is one reason why the independence assumption does not hold in general. Consider the program $\mathbb{C}_1$:

$l_1 : \mathtt{new}(x);$
$l_2 : \mathtt{new}(y);$
$l_3 : \mathtt{dispose}(x);$
$l_4 : \mathtt{if}(x = y)\mathtt{then}\{z := 0\}\mathtt{else}\{z := 1\};$

At l_4, we have the $x \neq y$ because x and y cannot be allocated the same heap cell, so the original program never sets $z := 0$. Now statements l_2 and l_3 are independent in that they access separate heap cells and variables, but reordering them may allow x and y to be equal if the cell allocated in l_1 is re-used for l_2. This means that the optimized program will possibly set $z := 0$, and so new behaviour can result from optimizations involving allocation and deallocation.

We first note that it is a widely accepted standard that programs should not read the values of pointers that are not allocated [16], and it may hence be argued that the above program is not an 'acceptable' program in the first place.

The following example shows that it is not simply a matter of disallowing programs that read dangling pointers[1]. Consider the program $\mathbb{C}_2$:

$l_1 : \mathtt{new}(x);$
$l_2 : \mathtt{new}(y);$
$l_3 : f := x;$
$l_4 : \mathtt{if}(f = y)\mathtt{then}\{z := 0\}\mathtt{else}\{z := 1\};$
$l_5 : \mathtt{dispose}(x);$
$l_6 : \mathtt{dispose}(y);$

In this case it is possible to optimize such that the statement sequence l_1, l_3, l_5 is executed even before y is allocated in l_2. Thus again, y may get the same address as x, and so the true branch in l_4 can fire in the optimization but not in the original program. So the problem exists even in programs that do not read values of dangling pointers, and which we cannot disallow. However, notice that in the $\mathbb{C}_2$ optimization we are still reading the value of f after assigning it to x and disposing x. Thus we need to somehow prevent optimizations that break the 'no dangling reads' rule.

Conceptually, we propose that the *footprint* of statements that read pointer variables should include the heap cell that the pointer is addressing. Thus a program that reads pointers that are not allocated, such as $\mathbb{C}_1$, will be considered to be a *faulting* program, just like one that dereferences a pointer that is not allocated. In the case of $\mathbb{C}_2$, the program itself is not faulting, but the footprint of l_4 in the dependency analysis will now include the heap allocated at f and y. This will prevent the illegal optimization because f and x address the same heap, and so the dispose command l_5 depends

[1] Note that we do not mean *dereferencing* dangling pointers, but even just looking at their value

$$
\begin{array}{l}
\{\mathtt{emp}\} \\
\quad l_1 : \mathtt{new}(x); \\
\{\langle x \mapsto 0\rangle_{\{l_1\}}\} \\
\quad l_2 : \mathtt{new}(y); \\
\{\langle x \mapsto 0\rangle_{\{l_1\}} * \langle y \mapsto 0\rangle_{\{l_2\}}\} \\
\quad l_3 : f := x; \\
\{f = x ¦ \langle x \mapsto 0\rangle_{\{l_1\}} * \langle y \mapsto 0\rangle_{\{l_2\}}\} \\
\{f = x ¦ \boxed{\langle f \mapsto 0\rangle_{\{l_1\}} * \langle y \mapsto 0\rangle_{\{l_2\}}}\} \\
\quad l_4 : \mathtt{if}(f = y)\mathtt{then}\{z := 0\}\mathtt{else}\{z := 1\}; \\
\left\{ \begin{array}{l} f \neq y \wedge z = 0 \wedge f = x ¦ \langle f \mapsto 0\rangle_{\{l_1,l_4\}} * \langle y \mapsto 0\rangle_{\{l_2,l_4\}}, \\ f = y \wedge z = 1 \wedge f = x ¦ \langle f \mapsto 0\rangle_{\{l_1,l_4\}} * \langle y \mapsto 0\rangle_{\{l_2,l_4\}} \end{array} \right\} \\
\{f \neq y \wedge z = 0 \wedge f = x ¦ \boxed{\langle x \mapsto 0\rangle_{\{l_1,l_4\}}} * \langle y \mapsto 0\rangle_{\{l_2,l_4\}}\} \\
\quad l_5 : \mathtt{dispose}(x); \\
\{f \neq y \wedge z = 0 \wedge f = x ¦ \boxed{\langle y \mapsto 0\rangle_{\{l_2,l_4\}}}\} \\
\quad l_6 : \mathtt{dispose}(y); \\
\{f \neq y \wedge z = 0 \wedge f = x\}
\end{array}
$$

Figure 7. Dependency analysis for $\mathbb{C}_2$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto F\rangle_L}{x = E[x'/x] \wedge (\Pi ¦ \Lambda * \langle E \mapsto F\rangle_{L \cup \{l\}})[x'/x]} \; l : x := E, x' \mathit{fresh}$$

$$\frac{\Pi ¦ \Lambda}{x = 0 \wedge (\Pi ¦ \Lambda)[x'/x]} \; l : x := E, \Pi ¦ \Lambda \vdash E = 0, x' \mathit{fresh}$$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto E'\rangle_L * \langle F \mapsto F'\rangle_{L'}}{\Pi ¦ \Lambda * \langle E \mapsto F\rangle_{L \cup \{l\}} * \langle F \mapsto F'\rangle_{L' \cup \{l\}}} \; l : [E] := F$$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto E'\rangle_L}{\Pi ¦ \Lambda * \langle E \mapsto F\rangle_{L \cup \{l\}}} \; l : [E] := F, \Pi ¦ \Lambda * \langle E \mapsto E'\rangle_L \vdash F = 0$$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto F\rangle_L * \langle F \mapsto F'\rangle_{L'}}{x = F[x'/x] \wedge (\Pi ¦ \Lambda * \langle E \mapsto F\rangle_{L \cup \{l\}} * \langle F \mapsto F'\rangle_{L' \cup \{l\}})[x'/x]} \; \dagger$$

$$\dagger \; l : x := [E], x' \; \mathit{fresh}$$

$$\frac{\Pi ¦ \Lambda * \langle E \mapsto F\rangle_L}{x = F[x'/x] \wedge (\Pi ¦ \Lambda * \langle E \mapsto F\rangle_{L \cup \{l\}})[x'/x]} \; \dagger$$

$$\dagger \; l : x := [E], x' \; \mathit{fresh}, \Pi ¦ \Lambda * \langle E \mapsto F\rangle_L \vdash F = 0$$

Figure 8. Assignment, mutation and lookup with increased footprint that includes the heap of the pointer that is being read

on l_4 and cannot be moved above it, which is also clearly what one would expect of any execution of the program.

Formally, we need to alter the dependency analysis of the last section to increase the footprint of commands that read pointer variables. Figure 8 shows the rules for the assignment, mutate and lookup commands which now require that if the pointer that is read is not null, then it must be allocated in the pre-state, and the label of the command is also added to the heap of the read pointer in the post-state. If the pointer to be read is null, then we have the usual execution rules from the previous section. We use similar case splits on pointers in the guards of conditionals and while loops, so that non-null pointers in the guards must be allocated in the pre-state and are part of the access footprint of the whole composite command. For example, the dependency analysis for program $\mathbb{C}_2$ is shown in Figure 7 (with boxed footprints). The footprint set of l_5 contains l_4, and so there is a dependency from l_5 to l_4 which prevents the incorrect optimization for $\mathbb{C}_2$ described earlier. We formally show soundness in section 4.3.

4.2 Non-terminating executions

The other problem with the independence assumption is that it does not account for non-terminating executions. For example, consider the program $\mathbb{C}_3$

$$
\begin{array}{l}
l_1 : \mathtt{while}(\mathtt{true})\{\mathtt{skip};\} \\
l_2 : [x] := 0;
\end{array}
$$

This program is safe but non-terminating on an initially empty heap, because of the infinite loop in l_1. In this case l_1 and l_2 do not access any common heap locations or variables in any possible execution of the program, and are hence independent by definition. But if we use the independence assumption to reorder l_1 and l_2, then the optimized program will be unsafe since it will access the unallocated cell at x. Also notice that it is not just a matter of *dead code* becoming live in the optimization, as can be seen in the program $\mathbb{C}_4$:

$$
\begin{array}{l}
l_1 : \mathtt{if}(y \geq 0)\mathtt{then}\{\mathtt{new}(x);\} \\
l_2 : \mathtt{while}(y \neq 0)\{y --;\} \\
l_3 : [x] := 0;
\end{array}
$$

If initially $y \geq 0$ then the program is safe and terminating and l_3 is executed. If $y < 0$, then the program does not terminate, but moving l_3 above l_2 will cause a memory fault.

We can address this problem using methods from [4], synthesizing conditions to underapproximate weakest preconditions. Note that if we can statically prove termination (using *e.g.* [6]), then $W = \mathtt{true}$. For any given sequential block $c = l_1 : c_1; \ldots; l_n : c_n$, we can synthesize an underapproximation to termination:

$$W = WP(c, \mathit{true})$$

and perform the optimization conditionally:

$$\mathtt{if}(W)\mathtt{then}\{c\}\mathtt{else}\{c\}$$

As an example, in the case of $\mathbb{C}_4$ we will get

$$\mathtt{if}(y \geq 0)\mathtt{then}\{\mathbb{C}_4\}\mathtt{else}\{\mathbb{C}_4\}$$

and safely reorder l_3 and l_2 in the true branch of the conditional.

Note that, while we can optimize non-terminating programs, we only perform the optimizations inside the bodies of loops, not outside (we check that the pre-state guarentees termination using an underapproximation to the weakest precondition). As an example consider the program in Figure 1, which has a non-terminating outer while loop. In this case the sequential block containing the outer loop will have weakest precondition false, so no optimizations will be applied at this level. But when the analysis is applied to the sequential block which is the body of this outer loop, then every trace is finite since it is a trace of a single iteration of the infinite loop. If the body happens to contain an infinite loop, then again the weakest precondition of the body will avoid optimizing any infinite traces. For this reason, when proving soundness, we will use a partial correctness semantics that does not consider infinite traces for loops. We need only consider finite traces from preconditions, thus modeling the execution within loop bodies.

4.3 Soundness

In this section we demonstrate soundness of the analysis using an action trace semantics of commands [2]. Traces are sequential composition of atomic actions, where atomic actions are primitive commands or assume statements, the formal semantics of which is defined below. The trace sets of programs are shown in Figure 9. For a sequential block $i_1 : c_1; \ldots; i_n : c_n$, we label all the atomic actions in all traces of c_k with the label i_k, for $1 \leq k \leq n$. Thus every trace of the sequential block is of the form $(i_1 : \tau_1); \ldots; (i_n : \tau_n)$, where $i_k : \tau_k$ is the trace τ_k in which every atomic action has the

$$T(a) = \{a\}$$
$$T(c_1; c_2) = \{\tau_1; \tau_2 \mid \tau_1 \in T(c_1), \tau_2 \in T(c_2)\}$$
$$T(\texttt{if}\ b\ c_1\ c_2) = \{\texttt{assume}(b); \tau_1 \mid \tau_1 \in T(c_1)\} \cup \{\texttt{assume}(\neg b); \tau_2 \mid \tau_2 \in T(c_2)\}$$
$$T(\texttt{while}\ b\ c) = (\texttt{assume}(b); T(c))^*; \texttt{assume}(\neg b)$$
where $(.)^*$ is Kleene-star (iterated ;)

Figure 9. Action trace semantics of commands

label i_k. Our aim is to show that any reordering of atomic actions in a trace under the dependencies determined by our analysis will produce equivalent output states to the original trace.

For the purposes of proving soundness, we work with a refined storage model which distinguishes whether a location has been disposed or not. We do this by extending the set of values with the set of *disposed* locations, $\texttt{Loc}_d = \{l^d \mid l \in \texttt{Loc}\}$. The idea is that when a location l is disposed by the dispose command, then every variable and cell that was pointing to l will now point to l^d to indicate that the variable or cell is holding a disposed location. Thus we have $\texttt{Val} = \texttt{Loc} \cup \{0\} \cup \texttt{Loc}_d$. Heaps and Stacks are as defined in Section 3 but States have a *well-formedness* constraint that all variables and cells either point to null, a disposed location or a location that is allocated on the heap. Thus every $(s, h) \in \texttt{States}$ is such that

$$\forall x \in \texttt{Var}.\ s(x) = 0 \vee s(x) \in \texttt{Loc}_d \vee s(x) \in dom(h)$$
$$\forall l \in dom(l).\ h(l) = 0 \vee h(l) \in \texttt{Loc}_d \vee h(l) \in dom(h)$$

Semantically, the primitive actions correspond to total functions that are of the form $\texttt{States} \rightarrow \mathcal{P}(\texttt{States})^\top$. The $\top$ element represents a faulting execution, that is, dereferencing a null pointer or an unallocated region of the heap, as well as the reading of dangling pointers, as we described in section 4.1.

The atomic actions and their denotational semantics are given in Figure 10. The semantics records the label of the action in the label set of the heap cells that the action accesses. In the case of dispose we substitute the disposed location in all the variables and cells that were referring to the cell that is disposed. The expression $(s, h)[l^d/l]$ is the state (s', h') defined as:

$$s'(x) = \begin{cases} l^d & \text{if } s(x) = l \\ s(x) & \text{otherwise} \end{cases} \qquad h'(l') = \begin{cases} l^d & \text{if } h'(l') = l \\ h'(l') & \text{otherwise} \end{cases}$$

The assume statements filter out all the states that satisfy the boolean condition, but fault if the expressions being read are not allocated locations. Similar faulting behaviour is shown by the assignment, lookup and mutation if the locations they read are not allocated in the heap, and this constraint ensures that disposed locations are never read in any safe execution.

Figure 10 also defines the *access label set* of every atomic action. For any atomic action $(i : a)$ acting on a state s, h, the access label set, $acc(i : a, s, h)$, is the set of labels in the part of h that is accessed by $i : a$ when $i : a$ is executed on the state s, h. Thus the access set of an action $i : a$ indicates all the previous actions that accessed the heap that the $i : a$ is accessing.

The semantics of an action trace is given by the sequential composition of its actions, defined as $[\![(i : a); (i' : a')]\!](s, h)$

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

We now define the notion of dependency between two actions in a trace, with respect to a given initial state.

DEFINITION 2 (Dependency). *Assume we have an action trace*

$$\tau = (i_1 : a_1); \ldots ; (i_n : a_n)$$

For any two actions $(i_u : a_u)$ and $(i_v : a_v)$ in τ such that $u < v$, there is a heap carried dependency between the two actions from an initial state (s, h), written

$$hdep(i_u : a_u, i_v : a_v, \tau, s, h)$$

iff $[\![(i_1 : a_1); \ldots ; (i_{v-1} : a_{v-1})]\!](s, h) = S$ and for some $(s', h') \in S$, we have $i_u \in acc(i_v : a_v, s', h')$.

We write $dep(i_u : a_u, i_v : a_v, \tau, s, h)$ if there is either a heap carried dependency or the two actions access common stack variables.

For a set of states S, we shall write $dep(i : a, i' : a', \tau, S)$ if there is some $(s, h) \in S$ such that $dep(i : a, i' : a', \tau, s, h)$, and similarly for $hdep$.

LEMMA 1. *The dependency detection algorithm from Figure 5 soundly determines heap carried dependencies, that is, for a sequential block c, if there is a dependency $hdep(i_u : a_u, i_v : a_v, \tau, s, h)$ between any two actions $(i_u : a_u)$ and $(i_v : a_v)$ in any trace τ of c for some (s, h) in the precondition, then $i_u \in deps(i_v)$ in the result of the algorithm.*

Our ultimate goal is to show soundness of the *transformations* made using this dependence information. That is, we have to show that starting from the same initial state, a non-diverging trace that is reordered under dependencies between atomic actions produces output states which are in some sense equivalent to the output of the original trace (only non-diverging traces represent real executions). This notion of equivalence is not always equality of the set of output states. For example, the program

$$c = \texttt{new}(x); \texttt{new}(y); \texttt{dispose}(x);$$

may be safely transformed to

$$c' = \texttt{new}(x); \texttt{dispose}(x); \texttt{new}(y);$$

In this case c' may produce a state in which x was allocated the same location as y, but c cannot give such an outcome. However, we observe that such differences relate to the specifics of the allocation behaviour of the program, and so the notion of equivalence we shall guarantee is that of equivalence up to *permutation of allocated locations*.

DEFINITION 3 (Structural equivalence). *We then define two states (s, h) and (s', h') to be structurally equivalent, written $(s, h) \simeq (s', h')$, iff $|dom(h)| = |dom(h')|$ and there exists a bijection $\pi : dom(h) \rightarrow dom(h')$ such that for all $l \in dom(h)$ and for all $x \in \texttt{Var}$,*

$$h(l) = (l', L) \wedge l' \in \texttt{Loc}_d \Rightarrow h'(\pi(l)) = (l'', L) \wedge l'' \in \texttt{Loc}_d$$
$$h(l) = (l', L) \wedge l' \notin \texttt{Loc}_d \Rightarrow h'(\pi(l)) = (\pi(l'), L)$$
$$h(l) = (0, L) \Rightarrow h'(\pi(l)) = (0, L)$$
$$s(x) \in \texttt{Loc}_d \Rightarrow s'(x) \in \texttt{Loc}_d$$
$$s(x) \notin \texttt{Loc}_d \Rightarrow s'(x) = \pi(s(x))$$
$$s(x) = 0 \Rightarrow s'(x) = 0$$

We may write $\pi(s, h) = (s', h')$ when the two states are related by permutation π.

LEMMA 2. *Structural equivalence, $\simeq$, is an equivalence relation. We write $[s, h]_\simeq$ for the equivalence class of s, h. For a set of states S, we write $[S]_\simeq$ to denote the union of the equivalence classes of all states in S.*

LEMMA 3 (Access). *For every atomic action $i : a$ and state s, h we have $acc(i : a, s, h) = acc(i : a, s_1, h_1)$ for every $(s_1, h_1) \in [s, h]_\simeq$.*

$$i : x := E \quad : \quad \begin{cases} \{s[x \mapsto 0], h\}, \emptyset & \text{if } \llbracket E \rrbracket s = 0 \\ \{s[x \mapsto l], h[l \mapsto (l', L \cup \{i\})]\}, L & \text{if } \llbracket E \rrbracket s = l \text{ and } h(l) = (l', L) \\ \top, \mathit{undef} & \text{otherwise} \end{cases}$$

$$i : x := [E] \quad : \quad \begin{cases} \{s[x \mapsto 0], h[l \mapsto (0, L \cup \{i\})]\}, L & \text{if } \llbracket E \rrbracket s = l, h(l) = (0, L) \\ \{s[x \mapsto l_1], h[l \mapsto (l_1, L_1 \cup \{i\})][l_1 \mapsto (l_2, L_2 \cup \{i\})]\}, L_1 \cup L_2 & \text{if } \llbracket E \rrbracket s = l, h(l) = (l_1, L_1) \text{ and } h(l_1) = (l_2, L_2) \\ \top, \mathit{undef} & \text{otherwise} \end{cases}$$

$$i : [E_1] := E_2 \quad : \quad \begin{cases} \{s, h[l \mapsto (0, L \cup \{i\})]\}, L & \text{if } \llbracket E_1 \rrbracket s = l, \llbracket E_2 \rrbracket s = 0 \text{ and } h(l) = (l', L) \\ \{s, h[l \mapsto (l_1, L_1 \cup \{i\})][l_1 \mapsto (l'', L_2 \cup \{i\})]\}, L_1 \cup L_2 & \text{if } \llbracket E_1 \rrbracket s = l, \llbracket E_2 \rrbracket s = l_1, \text{and } h(l) = (l', L_1) \text{ and } h(l_1) = (l'', L_2) \\ \top, \mathit{undef} & \text{otherwise} \end{cases}$$

$$i : \mathtt{new}(x) \quad : \quad \{s[x \rightarrow l], h * l \mapsto (0, \{i\}) \mid l \in \mathtt{Loc} \backslash \mathit{dom}(h)\}, \emptyset$$

$$i : \mathtt{dispose}(E) \quad : \quad \begin{cases} \{(s, h')[l^d / l]\}, L & \text{if } \llbracket E \rrbracket s = l, h = h' * l \mapsto (l', L) \\ \top, \mathit{undef} & \text{otherwise} \end{cases}$$

$$i : \mathtt{assume}(E_1 = E_2) \quad : \quad \begin{cases} \{s, h\}, \emptyset & \text{if } \llbracket E_1 \rrbracket s = 0, \llbracket E_2 \rrbracket s = 0 \\ \{s, h[l \mapsto (l', L \cup \{i\})]\}, L & \text{if } \llbracket E_1 \rrbracket s = l, \llbracket E_2 \rrbracket s = l, h(l) = (l', L) \\ \emptyset, L \cup L' & \text{if } \llbracket E_1 \rrbracket s = l, \llbracket E_2 \rrbracket s = l' \text{ and } h = h' * l \mapsto (l_1, L) * l' \mapsto (l_2, L') \\ \emptyset, L & \text{if } \llbracket E_1 \rrbracket s = l, \llbracket E_2 \rrbracket s = 0 \text{ and } h = h' * l \mapsto (l', L) \\ \emptyset, L & \text{if } \llbracket E_1 \rrbracket s = 0, \llbracket E_2 \rrbracket s = l \text{ and } h = h' * l \mapsto (l', L) \\ \top, \mathit{undef} & \text{otherwise} \end{cases}$$

$$i : \mathtt{assume}(E_1 \neq E_2) \quad : \quad \begin{cases} \emptyset, \emptyset & \text{if } \llbracket E_1 \rrbracket s = 0, \llbracket E_2 \rrbracket s = 0 \\ \emptyset, L & \text{if } \llbracket E_1 \rrbracket s = l, \llbracket E_2 \rrbracket s = l, h(l) = (l', L) \\ \{s, h' * l \mapsto (l_1, L \cup \{i\}) * l' \mapsto (l_2, L' \cup \{i\})\}, L \cup L' & \text{if } \llbracket E_1 \rrbracket s = l, \llbracket E_2 \rrbracket s = l' \text{ and } h = h' * l \mapsto (l_1, L) * l' \mapsto (l_2, L') \\ \{s, h[l \mapsto (l', L \cup \{i\})]\}, L & \text{if } \llbracket E_1 \rrbracket s = l, \llbracket E_2 \rrbracket s = 0 \text{ and } h = h' * l \mapsto (l', L) \\ \{s, h[l \mapsto (l', L \cup \{i\})]\}, L & \text{if } \llbracket E_1 \rrbracket s = 0, \llbracket E_2 \rrbracket s = l \text{ and } h = h' * l \mapsto (l', L) \\ \top, \mathit{undef} & \text{otherwise} \end{cases}$$

Figure 10. Denotational semantics and access labels sets of atomic actions. For every labelled atomic action $i : a$, the figure shows $\llbracket i : a \rrbracket(s, h)$, $acc(i : a, s, h)$ for a stack and heap state s, h.

LEMMA 4 (Closure). *If $\llbracket \tau \rrbracket(s, h) = S$ then all states in S are structurally equivalent. We also have $\llbracket \tau \rrbracket([s, h]_{\simeq}) \subseteq [S]_{\simeq}$ and if $S \neq \emptyset$ then $\llbracket \tau \rrbracket([s, h]_{\simeq}) \neq \emptyset$.*

LEMMA 5 (Reordering). *Let τ be a trace and $(i : a)$ an atomic action. Assume we have $\llbracket \tau; (i : a) \rrbracket(s, h) = S$, $S \neq \emptyset$ and $\neg dep(i' : a', i : a, \tau; (i : a), s, h)$ for all actions $(i' : a')$ in τ. We then have $\llbracket (i : a); \tau \rrbracket(s, h) \subseteq [S]_{\simeq}$ and $\llbracket (i : a); \tau \rrbracket(s, h) \neq \emptyset$.*

THEOREM 6 (Soundness). *Let C be a sequential block to which the dependency detection algorithm (Figure 5) has been applied for a certain precondition. Let s, h be a state in the precondition and let τ be a trace of C such that $\llbracket \tau \rrbracket(s, h) = S$ and $S \neq \emptyset$. If τ' is any reordering of actions in τ respecting computed dependencies, then we have $\llbracket \tau' \rrbracket(s, h) \subseteq [S]_{\simeq}$ and $\llbracket \tau' \rrbracket(s, h) \neq \emptyset$.*

5. Experimental evaluation

The preceding sections have described a method for using a shape analysis based on separation logic to determine heap-carried dependencies. In this section, we describe how this analysis can be used when synthesizing hardware. We also discuss the outcome of our preliminary experimental evaluation.

5.1 Implementation

Section 3 described a method of associating with each label l a set of labels L such that L is an over-approximation of the heap-carried data dependencies of the command at l. Such information specifies a partial ordering among commands and indicates that the command at l can not be executed before any command at $l' \in L$. To obtain the full set of data dependencies, we need to augment this information with the dependencies between instructions that arise due to interaction between stack variables. For example, in the code below, the command at l_3 is data-dependent on the command at l_1.

$$l_1 : x := 3; \;\; l_2 : y := 4; \;\; l_3 : z := x + 2;$$

In addition to these data dependencies, we also have control dependencies that arise whenever the execution of one command is dependent on the evaluation of another command, as occurs in conditional branches. In the program below, l_2 and l_3 are control-dependent on l_1 while l_4 is not, since l_4 is evaluated regardless of which branch is taken.

$$l_1 : \text{if}(x < 5) \text{ then } l_2 : y = 0 \text{ else } l_3 : y = 1;$$
$$l_4 : z = x;$$

Our implementation is based on the shape analysis tool [18]. Our tool uses known techniques for exploiting dependency information for instruction re-ordering and parallelization: [9] gives algorithms for taking sequential code, computing control dependencies, and combining this with data dependencies to obtain a *program dependence graph*. Such a graph can then be transformed into optimized sequential code using techniques adapted from [26, 11, 3]. The computed dependencies are then passed to our custom C to VHDL compiler for programs with heap. The compiler adopts the approach of [7] to introduce parallelism into to the design.

5.2 Experiments

We evaluated the benefit of the analysis using the following three examples:

Merge sort – This is our running example from Figure 1. The design has two input signals and one output signal. The implementation repeatedly inputs and sorts n elements through the first input signal and n elements through the second input signal. Using the merge sort it then combines the two sequences into one sorted sequence, which is then outputted. In our evaluation we used $n = 10$.

Priority queue – The design has one input signal and one output signal. The implementation repeatedly inputs and sorts and outputs n elements. In our evaluation we used $n = 10$.

Huffman encoder – This example implements a data structure for binary encoding of symbols. The design has three input signals and one output signal. The implementation repeatedly inputs n_1 symbols through the first input signal, their frequencies through the second input signal, and builds a Huffman encoder using this data. This is then followed by receiving n_2 symbols through the third input signal and producing their respective binary encodings

Design	C-ALUTs	Regs	M-Blocks	F-Max
Sequential Prio	5908	4745	4096	86.65 MHz
Parallel Prio	5867	4667	4096	82.14 MHz
Final Prio	5883	4697	4096	88.46 MHz
Sequential Merge	12951	5334	8192	76.62 MHz
Parallel Merge	12985	5298	8192	85.5 MHz
Final Merge	11850	9400	8192	79.29 MHz
Sequential Huffman	30186	16156	12288	77.92 MHz
Parallel Huffman	-	-	-	- MHz
Final Huffman	-	-	-	- MHz

Figure 11. Synthesis Results

Design	F-Cycles	N-Cycles
Sequential Prio	353	353
Parallel Prio	196	196
Final Prio	196	168
Sequential Merge	955	955
Parallel Merge	337	337
Final Merge	337	250
Sequential Huffman	1771	1771
Parallel Huffman	966	966
Final Huffman	972	342

Figure 12. Latency and Throughput Measurements

through the output signal. In our evaluation we used $n_1 = 10$ and $n_2 = 10$.

Three versions of VHDL designs were generated for each of the above examples. The *sequential* version models sequential execution of the original C program. The *parallel* version uses heap and stack dependencies to enable parallel execution of instructions. The *final* version extends the parallel version with memory localization and pipelining. Two pipeline stages were introduced in the Priority Queue and Merge Sort examples and four stages were introduced in the Huffman encoder example.

The generated circuits were synthesized for experimentation on an FPGA using Altera's Quartus II 9.0 tools. The synthesis phase involves transforming our generated VHDL descriptions into a circuit netlist (graph) which contains logical elements like combinational gates and registers for state information. The implementation phase involves mapping these components onto a specific FPGA chip by automatically places the nodes in the graph and then automatically routing wires between the nodes. The target FPGA architecture we use is Stratix III. The frequency results were obtained using a slow 1199mV 0C device model.

The table in Figure 11 gives the synthesis results for all nine VHDL designs considered here. The **C-ALUTs** column identifies the number of combinational four-input lookup tables used in the design. The **Regs** column identifies the number of registers flip-flops used in the design. The **M-Blocks** column identifies the number of bytes of memory blocks used in the design. The **F-Max** column identifies the maximum clocking frequency of the design.

Our synthesis results show very little variation in maximum clock frequency after parallelization and pipelining. This is largely due to the critical path being determined by the slowest statement in the finite state machine representation of the final circuit.

As of the writing of this document, the parallel and pipelined versions of the Huffman encoder circuit successfully synthesize from VHDL into gates, but the generated circuit is too large to fit onto a Stratix III FPGA. In the short term we believe it is possible to improve our synthesis flow to generate more efficient state machine representations that consume far fewer gates. One longer-term approach we are considering is to synthesize heap programs to Bluespec [21]. Bluespec generates Verilog output which is crafted to be well-aligned with commercial gates synthesis tools whereas we generate VHDL code without making such considerations.

A custom VHDL test bench was written for each design and ran in simulation to evaluate latency and throughput of our designs. The table in Figure 12 summarizes results of these measurements. The **F-Cycles** column identifies the number of clock cycles needed by the respective design to produce the first result. This measure represents the latency of the circuit. The **N-Cycles** column identifies the number of clock cycles needed by the respective design to produce the next result (assuming that inputs are always available). This measure represents the throughput of the circuit.

The table in Figure 12 tells us that using our analysis to introduce parallelism into a sequential design decreases latency by 45% for the Priority Queue and Huffman Encoder examples, and by 65% for the Merge Sort example, while the throughput is increased by 83% and 183% respectively. By manually applying pipelining and memory localization optimizations, the throughput gain grows to 110% for the Priority Queue example, 282% for the Merge Sort example, and 417% for the Huffman encoder example. As expected, the latency of the final version of the circuits matches that of the parallel version.

Let us note that the source of the measured throughput and latency gains of the parallel version over the sequential version are both stack and heap dependencies, with heap dependencies often making the difference; for example, the two input loops of the Merge Sort example could not be executed in parallel without computing fine heap dependencies. The source of the measured throughput gains of the final versions over the parallel versions are heap dependencies only. For our pipelined circuits, throughput is determined by the pipeline stage consuming the most clock cycles. The Priority Queue and Merge Sort examples both have one long and one short stage and thus the benefit of pipelining is modest. However, the Huffman Encoder example has four pipeline stages of comparable lengths. Consequently, the throughput gain for the Huffman encoder example is substantial.

6. Conclusion

In this paper we have described a separation logic based program analysis for indentifying heap-carried data dependencies between program statements, and shown how this allows us pipeline, parallelize, and localize memories when synthesizing circuits from C programs. In the future we might find the techniques described here useful when synthesizing hardware from lower-level hardware design languages such as VHDL or Verilog where heap-like structures are sometimes encoded explicitly in arrays. The techniques proposed here could also potentially be used for compilation for embedded systems. For example, ARM microprocessors provide support for software pipeling (*e.g.* predicated instructions). Finally, our support for memory localization allows for different memory management schemes to be used according to profiled behavior in the same program: this could have application both for C-to-gates synthesis, as well as compilation for embedded systems.

References

[1] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[2] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.

[3] J. M. P. Cardoso and H. C. Neto. Compilation for fpga-based reconfigurable hardware. *Design & Test of Computers, IEEE*, 20(2):65–75, 2003.

[4] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *CAV: International Conference on Computer Aided Verification*, 2008.

[5] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. Submitted. Available on Byron Cook's webpage, 2010.

[6] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[7] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[8] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[10] R. Ghiya, L. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive data structures. In *CC*, 1998.

[11] W. Gong. *Synthesizing sequential programs onto reconfigurable computing systems*. PhD thesis, Santa Barbara, CA, USA, 2007. Adviser-Kastner, Ryan.

[12] J. Hayman and G. Winskel. Independence and concurrent separation logic. In *LICS*, 2006.

[13] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for poiner variables. In *PLDI*, 1989.

[14] J. Hummel, L. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *PLDI*, 1994.

[15] C. Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In *SAS*, 2009.

[16] B. Kernighan and D. Ritchie. The c programming language. Prentice Hall, 1988.

[17] R. Lipton. Reduction: A method of proving properties of parallel programs. *CACM*, 18(12), 1975.

[18] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.

[19] M. Marron, D. Stefanovic, D. Kapur, and M. Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *LCPC*, 2008.

[20] J. Matthews and J. Launchbury. Elementary microarchitecture algebra. In *CAV*, 1999.

[21] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[22] P. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, 2004.

[23] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[24] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP*, pages 348–362, 2009.

[25] V. Sassone, M. Nielsen, and G. Winskel. Models for concurrency: Towards a classification. *TCS*, 170(1-2):297–348, 1996.

[26] J. Zeng, C. Soviani, and S. A. Edwards. Generating fast code from concurrent program dependence graphs. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 175–181, New York, NY, USA, 2004. ACM.

7. Appendix

LEMMA 1. The dependency detection algorithm from Figure 5 soundly determines heap carried dependencies, that is, for a sequential block c, if there is a dependency $hdep(i_u : a_u, i_v : a_v, \tau, s, h)$ between any two actions $(i_u : a_u)$ and $(i_v : a_v)$ in any trace τ of c for some (s, h) in the precondition, then $i_u \in deps(i_v)$ in the result of the algorithm.

Proof: We first need to check that the $getPost$ procedure from the algorithm correctly propagates command labels through the symbolic execution so that the label sets of the concrete heap at every stage satisfy the labelling on the symbolic state.

For each of the atomic commands we check that the symbolic execution rules for label propagation (shown in Figure 4 and updated in Figure 8) are sound. In the case of assignment, the first rule from figure 8 is

$$\frac{\Pi \mathbin{¦} \Lambda * \langle E \mapsto F\rangle_L}{x = E[x'/x] \wedge (\Pi \mathbin{¦} \Lambda * \langle E \mapsto F\rangle_{L \cup \{i\}})[x'/x]} \; i : x := E, x' fresh$$

In this case if a state $(s, h) \models \Pi \mathbin{¦} \Lambda * \langle E \mapsto F\rangle_L$, then $h = h' * l \mapsto (l', L')$ where $[\![E]\!]s = l$ and $L' \subseteq L$. By semantics of assignment from figure 10 we have that the concrete state after assignment is $s[x \mapsto l], h[l \mapsto (l', L' \cup \{i\})]$, which satisfies the symbolic post state

$$x = E[x'/x] \wedge (\Pi \mathbin{¦} \Lambda * \langle E \mapsto F\rangle_{L \cup \{i\}})[x'/x]$$

since $L' \cup \{i\} \subseteq L \cup \{i\}$.

Each of the other rules can be checked in a similar way using the semantics of commands from figure 10, and the rearrangement rules can be similarly checked to be sound implications. In the case of composite commands, it was shown in section 3.3 that frame inference determines the frame assertion Λ_F such that the pre-state and the post state of the command are of the form $\Pi \mathbin{¦} \Lambda * \Lambda_F$ and $\Pi' \mathbin{¦} \Lambda' * \Lambda_F$ respectively, so that the concrete heap h_F satisfying Λ_F in the pre-state is never accessed by the command. This means that no action in any action trace of the command accesses h_F, so the label sets in h_F are preserved through any trace execution. This is the only information that is used in the label propagation in the post-state since only the labels in Λ_F are preserved exactly in the post-state, and all the label sets in Λ' are conservatively assumed to be the union of all labels in Λ and the label of the composite command.

Next, we need to check that the $getFootprintLabels(c, \Pi \mathbin{¦} \Lambda)$ procedure overapproximates the access label set of the command c on all the states satisfying $\Pi \mathbin{¦} \Lambda$. For primitive commands, the algorithm applies the rearrangement rules and determines the footprint labels from the relevant points-to assertion. In the case of assignment, if we have the rearranged symbolic pre-state

$$\Pi \mathbin{¦} \Lambda * \langle E \mapsto F\rangle_L$$

then the computed footprint label set is L. A concrete state s, h satisfying this formula is of the form $s, h' * l \mapsto (l', L')$ where $[\![E]\!]s = l$ and $L' \subseteq L$. According to figure 10 the access label set of the command on this state is L', so the superset L is a valid overapproximation.

In the case of composite commands, for the pre-state $\Pi \mathbin{¦} \Lambda * \Lambda_F$, the footprint labels are taken to be the union of all the labels in Λ. If the concrete pre-state is $s, h * h_F$ and h satisfies Λ, then the set of labels in h is a subset of all the labels in Λ. Since h_F is never accessed by any action in any action trace of the composite command, the access label sets of any action in any action trace can only have labels from h or the label of the composite command itself. This is overapproximated by the labels in Λ and the label of the composite command. Note, however, that the algorithm actually does not include the label of the composite command in its computed footprint labels, as it is trivially true that there is a dependency from the command to itself. ■

LEMMA 2. Structural equivalence, $\simeq$, is an equivalence relation. We write $[s, h]_{\simeq}$ for the equivalence class of s, h. For a set of states

S, we write $[S]_{\simeq}$ to denote the union of the equivalence classes of all states in S.

Proof: Reflexivity follows by taking the relating bijection π to be the identity. For symmetry, if $(s,h) \simeq (s',h')$ and π is a relating bijection, then the inverse of π is a relating bijection for $(s',h') \simeq (s,h)$. For transitivity, if we have $(s,h) \simeq (s',h')$ and $(s',h') \simeq (s'',h'')$ related by bijections π and π' respectively, then the composition of π and π' is a relating bijection for $(s,h) \simeq (s'',h'')$. ■

LEMMA 3. For every atomic action $i : a$ and state s,h we have $acc(i:a,s,h) = acc(i:a,s_1,h_1)$ for every $(s_1,h_1) \in [s,h]_{\simeq}$.

Proof: For each primitive command $i : a$ we can check that if $(s_1,h_1) \simeq (s,h)$ then $acc(i:a,s,h) = acc(i:a,s_1,h_1)$. We assume that π is the relating permutation such that $\pi(s,h) = (s_1,h_1)$

For example, in the case of assignment, $i : x := E$, if $[\![E]\!]s = 0$ then $acc(i:a,s,h) = \emptyset$. By definition 3 we have $[\![E]\!]s_1 = 0$ and so $acc(i:a,s_1,h_1) = \emptyset = acc(i:a,s,h)$. If $[\![E]\!]s = l$ and $h(l) = (l',L)$ then by definition 3 we have $[\![E]\!]s_1 = \pi(l)$ and $h_1(\pi(l)) = (l'',L)$ for some l'', which gives $acc(i:a,s_1,h_1) = L = acc(i:a,s,h)$. Otherwise $[\![E]\!]s \in \mathtt{Loc}_d$, which means $[\![E]\!]s_1 \in \mathtt{Loc}_d$, and so both access sets are undefined. A similar argument applies to the other commands. ■

LEMMA 4. If $[\![\tau]\!](s,h) = S$ then all states in S are structurally equivalent. We also have $[\![\tau]\!]([s,h]_{\simeq}) \subseteq [S]_{\simeq}$ and if $S \neq \emptyset$ then $[\![\tau]\!]([s,h]_{\simeq}) \neq \emptyset$.

Proof: The proof is by induction on τ. We first check the base cases when τ is a primitive command $i : a$. For the first part of the lemma, every action other than allocation produces a single output state on input state s,h, and so the outputs are structurally equivalent. In the case of allocation, for input state s,h we may have any two output states $s, h * l_1 \rightarrow (0,\{i\})$ and $s, h * l_2 \rightarrow (0,\{i\})$. These two are related by the permutation π such that $\pi(l_1) = l_2$ and is the identity on $dom(h)$.

For the second part, Lemma 3 already implies that if $i : a$ is safe on (s,h) and $(s_1,h_1) \simeq (s,h)$ then $i : a$ is safe on (s_1,h_1). It remains to show that if $(s',h') \in [\![i:a]\!](s,h)$, then $[\![i:a]\!](s_1,h_1) \neq \emptyset$ and for all $(s_1',h_1') \in [\![i:a]\!](s_1,h_1)$, we have $(s_1',h_1') \simeq (s',h')$. If $[\![i:a]\!](s,h) = \emptyset$ then we should have $[\![i:a]\!](s_1,h_1) = \emptyset$.

The interesting cases are allocation, deallocation and the assume statements. In the case of allocation, let $(s',h') \in [\![i : \mathtt{new}(x)]\!](s,h)$ and $(s_1,h_1) \simeq (s,h)$. For any $(s_1',h_1') \in [\![i : \mathtt{new}(x)]\!](s_1',h_1')$, we have to show $(s_1,h_1) \simeq (s,h)$. We have $(s',h') = (s[x \rightarrow l], h * l \mapsto (0,\{i\}))$ for some $l \notin dom(h)$ and $(s_1',h_1') = (s_1[x \rightarrow l_1], h_1 * l_1 \mapsto (0,\{i\}))$ for some $l_1 \notin dom(h_1)$.

Let π be the permutation such that $\pi(s,h) = (s_1,h_1)$. We then define $\pi' : dom(h') \rightarrow dom(h_1')$ such that $\pi'(k) = \pi(k)$ if $k \in dom(h)$, and $\pi'(l) = l_1$. Then we have that $\pi'(s',h') = (s_1',h_1')$ and so they are structurally equivalent.

For dispose, let $[\![i : \mathtt{dispose}(E)]\!](s,h) = \{s',h'\}$ and $[\![E]\!]s = l$ and $h = h'' * l \mapsto (l',L)$. We have $(s',h') = (s,h'')[l^d/l]$. Now assume $(s_1,h_1) \simeq (s,h)$ and $\pi(s,h) = (s_1,h_1)$. By definition 3, we have $[\![E]\!]s_1 = \pi(l)$ and $h_1 = h_1'' * \pi(l) \mapsto (l_1',L)$ for some h_1'' and l_1'. This means that we have $[\![i : \mathtt{dispose}(E)]\!](s_1,h_1) = \{(s_1,h_1'')[l^d/l]\}$. Setting $\pi' : dom(h') \rightarrow dom(h_1')$ as the restriction of π to $dom(h)\backslash\{l\}$, we have $\pi'(s',h') = (s_1',h_1')$.

For $i : \mathtt{assume}(E_1 = E_2)$ on state (s,h) let $[\![E_1]\!]s = l$ and $[\![E_2]\!]s = l'$. Assume we have $\pi(s,h) = (s_1,h_1)$ and $[\![E_1]\!]s_1 = l_1$, $[\![E_2]\!]s_1 = l_1'$. By definition 3, we have l and l' are allocated in h and equal if and only if l_1 and l_1' are allocated in h_1 and equal. We also have $l = 0$ if and only if $l_1 = 0$ and $l' = 0$ if and only if $l_1' = 0$ and $l \in \mathtt{Loc}_d$ if and only if $l_1 \in \mathtt{Loc}_d$ and $l' \in \mathtt{Loc}_d$ if and only if $l_1' \in \mathtt{Loc}_d$. Thus the assume command diverges on (s,h) if and only if it diverges on (s_1,h_1). If it does not diverge then the output states are identical to the input, and so we have $\pi(s,h) = (s_1,h_1)$, which shows that the output states are structurally equivalent. A similar argument holds for the inequality assume command.

For assignment, mutation and lookup, it can be checked in a similar way that for structurally equivalent input states (s,h) and (s_1,h_1) related by a permutation π, the output states are also related by the same permutation π.

We now do the inductive case for arbitrary τ. So assume that we have $\tau = \tau';(i:a)$, $[\![\tau]\!](s,h) = S$ and that the lemma holds for τ' by the induction hypothesis. So let $[\![\tau']\!](s,h) = S_1$. For the first part of the lemma, by induction hypothesis we have that all states in S_1 are structurally equivalent, and so by the base case we get that all states in S are structurally equivalent, since $[\![i:a]\!](S_1) = S$.

For the second part of the lemma, we have $[\![\tau']\!]([s,h]_{\simeq}) = S_1' \subseteq [S_1]_{\simeq}$ by the induction hypothesis. We also have by the base case that $[\![i:a]\!]([S_1]_{\simeq}) \subseteq [S]_{\simeq}$ since $[\![i:a]\!](S_1) = S$, which gives $[\![i:a]\!](S_1') \subseteq [S]_{\simeq}$. So putting them together we get $[\![\tau]\!]([s,h]_{\simeq}) \subseteq [S]_{\simeq}$.

If $S \neq \emptyset$, then $S_1 \neq \emptyset$, which by induction hypothesis implies that $S_1' \neq \emptyset$. The only way that $(i:a)$ can diverge on S_1' is if it is an assume statement, and we saw in the proof of assume statements that structurally equivalent states have the same divergence behaviour. Hence, since $(i:a)$ does not diverge on S_1 and all states in S_1' are structurally equivalent to those in S_1, we have that $(i:a)$ does not diverge on S_1', and we get $[\![\tau]\!]([s,h]_{\simeq}) \neq \emptyset$.

■

LEMMA 5. Let τ a trace and $(i:a)$ an atomic action. Assume we have $[\![\tau;(i:a)]\!](s,h) = S$, $S \neq \emptyset$ and $\neg dep(i':a', i:a, \tau;(i:a), s, h)$ for all actions $(i':a')$ in τ. We then have $[\![(i:a);\tau]\!](s,h) \subseteq [S]_{\simeq}$ and $[\![(i:a);\tau]\!](s,h) \neq \emptyset$.

Proof: The proof is by induction on τ. For the base case we assume that τ is an atomic action $(i':a')$, and check all the commands individually. When $(i:a)$ and $(i':a')$ are not a combination of allocation and deallocation, we have that reordering preserves equality of states rather than just structural equivalence. That is, if $[\![(i':a') : (i:a)]\!](s,h) = S$ then $[\![(i:a) : (i':a')]\!](s,h) = S$. This follows by checking in each case that the access set of $(i:a)$ on the intermediate state $[\![i':a']\!](s,h)$ does not contain i', and so the two actions commute because they alter different parts of the heap and stack and leave the rest unchanged.

The interesting cases are the combination of allocation and deallocation. Say we have $[\![(i':\mathtt{new}(x));(i:\mathtt{dispose}(E))]\!](s,h) = S$ and $[\![E]\!]s = l$. In this case when we reorder we get $[\![(i:\mathtt{dispose}(E));(i':\mathtt{new}(x))]\!](s,h) = S \cup \{h' * l \rightarrow (0,\{i'\})\}$ where $h = h' * l \mapsto (l',L)$ for some l',L. The new state is structurally equivalent to every state in S because any state in S is of the form $h' * l_1 \rightarrow (0,\{i'\})$. The two states are related by a permutation π such that $\pi(l_1) = l$ and π is the identity on $dom(h')$.

The other case is when we have $[\![(i':\mathtt{dispose}(E));(i:\mathtt{new}(x))]\!](s,h) = S$ and $[\![E]\!]s = l$. In this case when we reorder we get $[\![(i:\mathtt{new}(x));(i':\mathtt{dispose}(E))]\!](s,h) = S' \subset S$. This is because l may have been allocated in S but it cannot be allocated in S' as it has not been deallocated yet. In this case structural equivalence is preserved since $S' \subset S$.

In all cases it can be checked that the access label set of $(i:a)$ on (s,h) is the same as the access set of $(i:a)$ on all states in $[\![i':a']\!](s,h)$.

For the inductive case, we assume that $\tau = \tau';(i':a')$ and that $[\![\tau';(i':a');(i:a)]\!](s,h) = S$ and $S \neq \emptyset$ and that the lemma holds for τ'. Let $[\![\tau']\!](s,h) = S'$. Since there is no

dependency between $(i : a)$ and $(i' : a')$ we have by the base case that $[\![\tau'; (i : a); (i' : a')]\!](s,h) \subseteq [S]_{\simeq}$ and $[\![\tau'; (i : a); (i' : a')]\!](s,h) \neq \emptyset$ and the access set of $(i : a)$ on S' is the same as its access set on $[\![i' : a']\!](S')$. Hence $(i : a)$ is independent of all actions in τ' from (s,h). So by induction hypothesis we have $[\![(i : a); \tau']\!](s,h) = S'' \subseteq [S''']_{\simeq}$ where $S''' = [\![\tau'; (i : a)]\!](s,h)$ and $S'' \neq \emptyset$. Since we have $[\![i' : a']\!](S''') \subseteq [S]_{\simeq}$, this gives $[\![(i : a); \tau]\!](s,h) \subseteq [S]_{\simeq}$. We also have $[\![(i : a); \tau]\!](s,h) \neq \emptyset$, because the only way it can diverge is if $(i' : a')$ is an assume statement, but the divergence behaviour of assume statements is the same on structurally equivalent states by proof of lemma 4, and S'' is structurally equivalent to S''' .
■

THEOREM 6. Let C be a sequential block to which the dependency detection algorithm (Figure 5) has been applied for a certain precondition. Let s, h be a state in the precondition and let τ be a trace of C such that $[\![\tau]\!](s,h) = S$ and $S \neq \emptyset$. If τ' is any reordering of actions in τ respecting computed dependencies, then we have $[\![\tau']\!](s,h) \subseteq [S]_{\simeq}$ and $[\![\tau']\!](s,h) \neq \emptyset$.

Proof: By Lemma 1 we have that the computed dependencies include all the valid dependencies in τ from state (s,h). Thus we show the result for any reordering τ' of τ respecting the dependencies between atomic actions in τ from state s, h. We show this by induction on τ. The base case is when τ is an atomic action, and in this case the result follows by lemma 5.

For the inductive case, assume that $\tau = \tau_1; (i : a)$ and that $[\![\tau_1]\!](s,h) = S_1$, where we know that $S_1 \neq \emptyset$ since $S \neq \emptyset$. Let $\tau' = \tau''; (i : a); \tau'''$ be the dependency respecting reordering of τ, where for all actions $(i' : a')$ in τ''', we have $\neg dep(i' : a', i : a, \tau, s, h)$. We then have that $\tau''; \tau'''$ is a dependency respecting reordering of τ_1, since for every pair of actions $(i_1 : a_1)$ and $(i_2 : a_2)$ in τ_1, there is a dependency $dep(i_1 : a_1, i_2 : a_2, \tau_1, s, h)$ iff there is a dependency $dep(i_1 : a_1, i_2 : a_2, \tau, s, h)$.

So by the induction hypothesis we get $[\![\tau''; \tau''']\!](s,h) = S_1' \subseteq [S_1]_{\simeq}$ and $S_1' \neq \emptyset$. By lemma 4, we have that all the states in S_1 are structurally equivalent, and so by lemma 3, we know that $(i : a)$ has the same access set L on all states in S_1' as it does on all states in S_1. This access set L does not contain the labels of any action in τ'''. This is because we had that $\tau' = \tau''; (i : a); \tau'''$ is a valid dependency respecting reordering of τ, and so by definition 2, none of the actions in τ''' have labels that are in L.

So now we have $[\![\tau''; \tau''']\!](s,h) = S_1'$ and that $(i : a)$ has access set L on all states in S_1', where L does not contain any labels of actions in τ'''. This means that in the trace $\tau''; \tau'''; (i : a)$ from state (s,h), there is no dependency between $(i : a)$ and any actions in τ'''. Hence by lemma 5, we have $[\![\tau''; (i : a); \tau''']\!](s,h) \subseteq [S]_{\simeq}$ and $[\![\tau''; (i : a); \tau''']\!](s,h) \neq \emptyset$.
■