Shape Analysis by Graph Decomposition

R. Manevich[1,⋆], J. Berdine[3], B. Cook[3], G. Ramalingam[2], and M. Sagiv[1]

[1] Tel Aviv University, {rumster,msagiv}@post.tau.ac.il
[2] Microsoft Research India, grama@microsoft.com
[3] Microsoft Research Cambridge, {bycook,jjb}@microsoft.com

Abstract. Programs commonly maintain multiple linked data structures. Correlations between multiple data structures may often be *non-existent or irrelevant to verifying that the program satisfies certain safety properties or invariants*. In this paper, we show how this *independence* between different (singly-linked) data structures can be utilized to perform shape analysis and verification more efficiently. We present a new abstraction based on decomposing graphs into sets of subgraphs, and show that, in practice, this new abstraction leads to very little loss of precision, while yielding substantial improvements to efficiency.

1 Introduction

We are interested in verifying that programs satisfy various safety properties (such as the absence of null dereferences, memory leaks, dangling pointer dereferences, etc.) and that they preserve various data structure invariants.

Many programs, such as web-servers, operating systems, network routers, etc., commonly maintain multiple linked data-structures in which data is added and removed throughout the program's execution. The Windows IEEE 1394 (firewire) device driver, for example, maintains separate cyclic linked lists that respectively store bus-reset request packets, data regarding CROM calls, data regarding addresses, and data regarding ISOCH transfers. These lists are updated throughout the driver's execution based on events that occur in the machine. Correlations between multiple data-structures in a program, such as those illustrated above, may often be *non-existent or irrelevant to the verification task of interest*. In this paper, we show how this *independence* between different data-structures can be utilized to perform verification more efficiently.

Many scalable heap abstractions typically maintain no correlation between different *points-to* facts (and can be loosely described as *independent attribute* abstractions in the sense of [7]). Such abstractions are, however, not precise enough to prove that programs preserve data structure invariants. More precise abstractions for the heap that use shape graphs to represent *complete* heaps [17], however, lead to exponential blowups in the state space.

In this paper, we focus on (possibly cyclic) singly-linked lists and introduce an approximation of the *full heap abstraction* presented in [13]. The new *graph*

⋆ This research was partially supported by the Clore Fellowship Programme. Part of this research was done during an internship at Microsoft Research India.

decomposition abstraction is based on a decomposition of (shape) graphs into sets of (shape) subgraphs (without maintaining correlations between different shape subgraphs). In our initial empirical evaluation, this abstraction produced results almost as precise as the full heap abstraction (producing just one false positive), while reducing the state space significantly, sometimes by exponential factors, leading to dramatic improvements to the performance of the analysis. We also hope that this abstraction will be amenable to abstraction refinement techniques (to handle the cases where correlations between subgraphs are necessary for verification), though that topic is beyond the scope of this paper.

One of the challenges in using a subgraph abstraction is the design of safe and precise transformers for statements. We show in this paper that the computation of the most precise transformer for the graph decomposition abstraction is FNP-complete.

We derive efficient, polynomial-time, transformers for our abstraction in several steps. We first use an observation by Distefano et al. [3] and show how the most precise transformer can be computed more efficiently (than the naive approach) by: (a) identifying *feasible combinations of subgraphs referred to by a statement*, (b) composing only them, (c) transforming the composed subgraphs, and (d) decomposing the resulting subgraphs. Next, we show that the transformers can be computed in polynomial time by omitting the feasibility check (which entails a possible loss in precision). Finally, we show that the resulting transformer can be implemented in an *incremental* fashion (i.e., in every iteration of the fixed point computation, the transformer reuses the results of the previous iteration).

We have developed a prototype implementation of the algorithm and compared the precision and efficiency (in terms of both time and space) of our new abstraction with that of the full heap abstraction over a standard suite of shape analysis benchmarks as well as on models of a couple of Windows device drivers. Our results show that the new analysis produces results as precise as the full heap-based analysis in almost all cases, but much more efficiently.

Outline. The rest of the paper is organized as follows. Sec. 2 gives a motivation for our analysis. Sec. 3 describes a concrete semantics for programs with linked lists and a full heap abstraction. Sec. 4 describes the graph decomposition abstraction. In Sec. 5 we develop efficient transformers for the graph decomposition abstraction. Sec. 6 presents experimental results and compares the full heap abstraction with the graph decomposition abstraction. Finally, in Sec. 7 we discuss related work.

2 Overview

In this section, we provide an informal overview of our approach. Later sections provide the formal details.

Fig. 1 shows a simple program that adds elements into independent lists: a list with a head object referenced by a variable `h1` and a tail object referenced

```
//@assume h1!=null && h1==t1 && h1.n==null && h2!=null && h2==t2 && h2.n==null
//@invariant Reach(h1,t1) && Reach(h2,t2) && DisjointLists(h1,h2)
EnqueueEvents() {
L1:  while (...) {
       List temp = new List(getEvent());
       if (nondet()) {
L2:      t1.n = temp;
L3:      t1 = temp;
       } else {
         t2.n = temp;
         t2 = temp;
}   }  }
```

Fig. 1. A program that enqueues events into one of two lists. `nondet()` returns either `true` or `false` non-deterministically

by a variable `t1`, and a list with a head object referenced by a variable `h2` and a tail object referenced by a variable `t2`. This example is used as the running example throughout the paper. The goal of the analysis is to prove that the data structure invariants are preserved in every iteration, i.e., at label `L1` variables `h1` and `t1` and variables `h2` and `t2` point to disjoint acyclic lists, and that the head and tail pointers point to the first and last objects in every list, respectively.

The shape analysis presented in [13] is able to verify the invariants by generating, at program label `L1`, the 9 abstract states shown in Fig. 2. These states represent the 3 possible states that each list can have: a) a list with one element, b) a list with two elements; and c) a list with more than two elements. This analysis uses a *full heap abstraction*: it does not take advantage of the fact that there is no interaction between the lists, and explores a state-space that contains all 9 possible combinations of cases $\{a, b, c\}$ for the two lists.

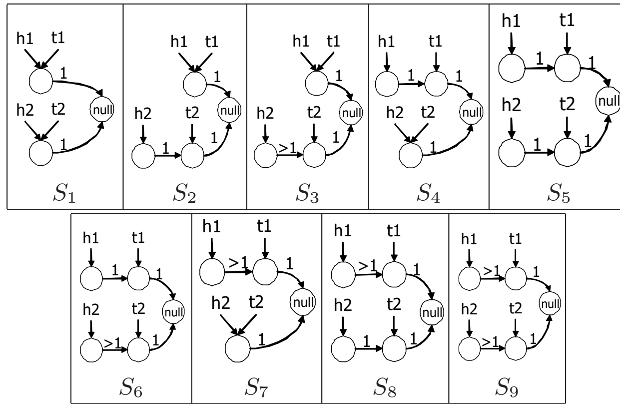


Fig. 2. Abstract states at program label `L1`, generated by an analysis of the program in Fig. 1 using a powerset abstraction. Edges labeled 1 indicate list segments of length 1, whereas edges labeled with >1 indicate list segments of lengths greater than 1

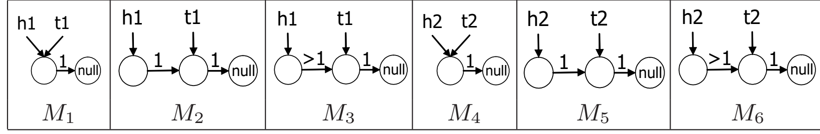


Fig. 3. Abstract states at program label `L1`, generated by an analysis of the program in Fig. 1 using the graph decomposition abstraction

The shape analysis using a *graph decomposition abstraction* presented in this paper, represents the properties of each list separately and generates, at program label `L1`, the 6 abstract states shown in Fig. 3. For a generalization of this program to k lists, the number of states generated at label `L1` by using a graph decomposition abstraction is $3 \times k$, compared to 3^k for an analysis using a full heap abstraction, which tracks correlations between properties of all k lists. In many programs, this exponential factor can be significant. Note that in cases where there is no *correlation* between the different lists, the new abstraction of the set of states is as precise as the full heap abstraction: e.g., Fig. 3 and Fig. 2 represent the same set of concrete states.

We note that in the presence of pointers, it is not easy to decompose the verification problem into a set of sub-problems to achieve similar benefits. For example, current (flow-insensitive) alias analyses would not be able to identify that the two lists are disjoint.

3 A Full Heap Abstraction for Lists

In this section, we describe the concrete semantics of programs manipulating singly-linked lists and a full heap abstraction for singly-linked lists.

A Simple Programming Language for Singly-Linked Lists. We now define a simple language and its concrete semantics. Our language has a single data type *List* (representing a singly-linked list) with a single reference field `n` and a data field, which we conservatively ignore.

There are five types of heap-manipulating statements: (1) `x=new List()`, (2) `x=null`, (3) `x=y`, (4) `x=y.n`, and (5) `x.n=y`. Control flow is achieved by using `goto` statements and `assume` statements of the form `assume(x==y)` and `assume(x!=y)`. For simplicity, we do not present a deallocation, `free(x)`, statement and use garbage collection instead. Our implementation supports memory deallocation, assertions, and detects (mis)use of dangling pointers.

Concrete States. Let $PVar$ be a set of variables of type *List*. A concrete program state is a triple $C \doteq (U^C, env^C, n^C)$ where U^C is the set of heap objects, an environment $env^C : PVar \cup \{null\} \to U^C$ maps program variables (and $null$) to heap objects, and $n^C : U^C \to U^C$, which represents the `n` field, maps heap objects to heap objects. Every concrete state includes a special object v_{null} such that $env(null) = v_{null}$.We denote the set of all concrete states by *States*.

Concrete Semantics. We associate a transition function $[\![st]\!]$ with every statement st in the program. Each statement st takes a concrete state C, and transforms it to a state $C' = [\![st]\!](C)$. The semantics of a statement is given by a pair $(condition, update)$ such that when the condition specified by *condition* holds the state is updated according to the assignments specified by *update*. The concrete semantics of program statements is shown in Tab. 1.

Table 1. Concrete semantics of program statements. Primed symbols denote post-execution values. We write x,y, and x' to mean $env(x)$, $env(y)$, and $env'(x)$, respectively

Statement	Condition	Update
`x=new List()`		$x' = v_{new}$, where v_{new} is a fresh List object $n' = \lambda\, v\, .\, (v = v_{new}\ ?\ null\ :\ n(v))$
`x=null`		$x' = null$
`x=y`		$x' = y$
`x=y.n`	$y \neq null$	$x' = n(y)$
`x.n=y`	$x \neq null$	$n' = \lambda\, v\, .\, (v = x\ ?\ y\ :\ n(v))$
`assume(x!=y)`	$x \neq y$	
`assume(x==y)`	$x = y$	

3.1 Abstracting List Segments

The abstraction is based on previous work on analysis of singly-linked lists [13]. The core concepts of the abstraction are *interruptions* and *uninterrupted list*. An object is an *interruption* if it is referenced by a variable (or *null*) or shared (i.e., has two or more predecessors). An uninterrupted list is a path delimited by two interruptions that does not contain interruptions other than the delimiters.

Definition 1 (Shape Graphs). *A shape graph* $G \doteq (V^G, E^G, env^G, len^G)$ *is a quadruple where* V^G *is a set of nodes,* E^G *is a set of edges,* $env^G : PVar \cup \{null\} \to V^G$ *maps variables (and null) to nodes, and* $len^G : E^G \to pathlen$ *assigns labels to edges. In this paper, we use* $pathlen \doteq \{1, >1\}$.[4]

We denote the set of shape graphs by SG_{PVar}, omitting the subscript if no confusion is likely, and define equality between shape graphs by isomorphism. We say that a variable `x` points to a node $v \in V^G$ if $env^G(\mathtt{x}) = v$.

We now describe how a concrete state $C \doteq (U^C, env^C, n^C)$ is abstracted into a shape graph $G \doteq (V^G, E^G, env^G, len^G)$ by the function $\beta^{FH} : States \to SG$. First, we remove any node in U^C that is not reachable from a (node pointed-to by a) program variable. Let $PtVar(C)$ be the set of objects pointed-to by

[4] The abstraction in [13] is more precise, since it uses the abstract lengths $\{1, 2, > 2\}$. We use the lengths $\{1, > 1\}$, which we found to be sufficiently precise, in practice.

some variable, and let $\mathit{Shared}(C)$ the set of heap-shared objects. We create a shape graph $\beta^{FH}(C) \doteq (V^G, E^G, env^G, len^G)$ where $V^G \doteq \mathit{PtVar}(C) \cup \mathit{Shared}(C)$, $E^G \doteq \{(u,v) \mid (u,\ldots,v)$ is an uninterrupted list$\}$, env^G restricts env^C to V^G, and $len^G(u,v)$ is `1` if the uninterrupted list from u to v has one edge and `>1` otherwise. The abstraction function α^{FH} is the point-wise extension of β^{FH} to sets of concrete states[5]. We say that a shape graph is *admissible* if it is in the image of β^{FH}.

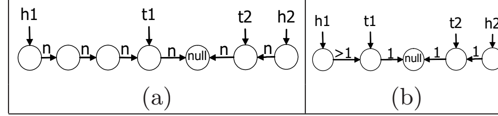


Fig. 4. (a) A concrete state, and (b) The abstraction of the state in (a)

Proposition 1. *A shape graph is admissible iff the following properties hold: (i) Every node has a single successor; (ii) Every node is pointed-to by a variable (or null) or is a shared node, and (iii) Every node is reachable from (a node pointed-to by) a variable.*

We use Prop. 1 to determine if a given graph is admissible in linear time and to conduct an efficient isomorphism test for two shape graphs in the image of the abstraction. It also provides a bound on the number of admissible shape graphs: $2^{5n^2+10n+8}$, where $n \doteq |PVar|$.

Example 1. Fig. 4(a) shows a concrete state that arises at program label `L1` and Fig. 4(b) shows the shape graph that represents it. □

Concretization. The function $\gamma^{FH} : SG \to 2^{States}$ returns the set of concrete states that a shape graph represents: $\gamma^{FH}(G) \doteq \{C \mid \beta^{FH}(C) = G\}$. We define the concretization of sets of shape graphs by using its point-wise extension. We now have the Galois Connection $\langle 2^{States}, \alpha^{FH}, \gamma^{FH}, 2^{SG}\rangle$.

Abstract Semantics. The most precise, a.k.a *best*, abstract transformer [2] of a statement is given by $[\![st]\!]^{\#} \doteq \alpha^{FH} \circ [\![st]\!] \circ \gamma^{FH}$. An efficient implementation of the most precise abstract transformer is shown in the full version [11].

4 A Graph Decomposition Abstraction for Lists

In this section, we introduce the abstraction that is the basis of our approach as an approximation of the abstraction shown in the previous section. We define

[5] In general, the point-wise extension of a function $f : D \to D$ is a function $f : 2^D \to 2^D$, defined by $f(S) \doteq \{f(s) \mid s \in S\}$. Similarly, the extension of a function $f : D \to 2^D$ is a function $f : 2^D \to 2^D$, defined by $f(S) \doteq \bigcup_{s \in S} f(s)$.

the domain we use—2^{ASSG}, the powerset of atomic shape subgraphs—as well as the abstraction and concretization functions between 2^{SG} and 2^{ASSG}.

4.1 The Abstract Domain of Shape Subgraphs

Intuitively, the graph decomposition abstraction works by decomposing a shape graph into a set of *shape subgraphs*. In principle, different graph decomposition strategies can be used to get different abstractions. However, in this paper, we focus on decomposing a shape graph into a set of subgraphs induced by its *(weakly-)connected components*. The motivation is that different weakly connected components mostly represent different "logical" lists (though a single list may occasionally be broken into multiple weakly connected components during a sequence of pointer manipulations) and we would like to use an abstraction that decouples the different logical lists. We will refer to an element of SG_{PVar} as a shape graph, and an element of SG_{Vars} for any $Vars \subseteq PVar$ as a shape subgraph. We denote the set of shape subgraphs by SSG and define $Vars(G)$ to be the set of variables that appear in G, i.e., mapped by env^G to some node.

4.2 Abstraction by Graph Decomposition

We now define the decomposition operation. Since our definition of shape graphs represents *null* using a special node, we identify connected components *after excluding the null node*. (Otherwise, all *null*-terminated lists, i.e. all acyclic lists, will end up in the same connected component.)

Definition 2 (Projection). *Given a shape subgraph* $G \doteq (V, E, env, len)$ *and a set of nodes* $W \subseteq V$, *the subgraph of* G *induced by* W, *denoted by* $G|_W$, *is the shape subgraph* (W, E', env', len'), *where* $E' \doteq E \cap (W \times W)$, $env' \doteq env \cap (Vars(G) \times W)$, *and* $len' \doteq len \cap (E' \times pathlen)$.

Definition 3 (Connected Component Decomposition). *For a shape subgraph* $G \doteq (V, E, env, len)$, *let* $R \doteq E'^*$ *be the reflexive, symmetric, transitive closure of the relation* $E' \doteq E \setminus \{(v_{null}, v), (v, v_{null}) \mid v \in V\}$. *That is,* R *does not represent paths going through null. Let* $[R]$ *be the set of equivalence classes of* R. *The connected component decomposition of* G *is given by*

$$Components(G) \doteq \{G|_{C'} \mid C' = C \cup \{v_{null}\}, C \in [R]\} \ .$$

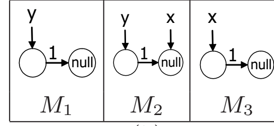
Example 2. Referring to Fig. 2 and Fig. 3, we have $Components(S_2) = \{M_1, M_5\}$.

Abstracting Away Null-value Correlations. The decomposition *Components* manages to decouple distinct lists in a shape graph. However, it fails to decouple lists from null-valued variables.

Example 3. Consider the code fragment shown in Fig. 5(a) and the shape subgraphs arising after `y=new List()`. `y` points to a list (with one cell), while `x` is

```
if (?) x = new List() else x = null;
y = new List();
```

(a)



(b)

Fig. 5. (a) A code fragment; and (b) Shape subgraphs arising after executing `y=new List()`. M_1: `y` points to a list and `x` is not null, M_2: `y` points to a list and `x` is null; and M_3: `x` points to a list and `y` is not null

null or points to another list (with one cell). Unfortunately, the `y` list will be represented by two shape subgraphs in the abstraction, one corresponding to the case that `x` is *null* (M_2) and one corresponding to the case that `x` is not *null* (M_1). If a number of variables can be optionally null, this can lead to an exponential blowup in the representation of other lists! Our preliminary investigations show that this kind of exponential blow-up can happen in practice. □

The problem is the occurrence of shape subgraphs that are isomorphic except for the *null* variables. We therefore define a coarser abstraction by decomposing the set of variables that point to the *null* node. To perform this further decomposition, we define the following operations:

- $\mathit{nullvars} : \mathit{SSG} \to 2^{\mathit{PVar}}$ returns the set of variables that point to *null* in a shape subgraph.
- $\mathit{unmap} : \mathit{SSG} \times 2^{\mathit{PVar}} \to \mathit{SSG}$ removes the mapping of the specified variables from the environment of a shape subgraph.
- $\mathit{DecomposeNullVars} : \mathit{SSG} \to 2^{\mathit{SSG}}$ takes a shape subgraph and returns: (a) the given subgraph without the null variables, and (b) one shape subgraph for every null variable, which contains just the null node and the variable:

$$\begin{aligned}\mathit{DecomposeNullVars}(G) \doteq \{\mathit{unmap}(G, \mathit{nullvars}(G))\} \cup \\ \{\mathit{unmap}(G|_{v_{null}}, \mathit{Vars}(G) \setminus \{\mathit{var}\} \mid \mathit{var} \in \mathit{nullvars}(G)\} \ .\end{aligned}$$

In the sequel, we use the point-wise extension of *DecomposeNullVars*.

We define the set *ASSG* of *atomic* shape subgraphs to be the set of subgraphs that consist of either a single connected component or a single *null*-variable fact (i.e., a single variable pointing to the *null* node). Non-atomic shape subgraphs correspond to conjunctions of atomic shape subgraphs and are useful intermediaries during concretization and while computing transformers.

The abstraction function $\beta^{GD} : \mathit{SG} \to 2^{\mathit{ASSG}}$ is given by

$$\beta^{GD}(G) \doteq \mathit{DecomposeNullVars}(\mathit{Components}(G)) \ .$$

The function $\alpha^{GD} : 2^{\mathit{SG}} \to 2^{\mathit{ASSG}}$ is the point-wise extension of β^{GD}. Thus, $\mathit{ASSG} = \alpha^{GD}(\mathit{SG})$ is the set of shape subgraphs in the image of the abstraction.

Note: We can extend the decomposition to avoid exponential blowups created by different sets of variables pointing to the same (non-*null*) node. However, we

believe that such correlations are significant for shape analysis (as they capture different states of a single list) and abstracting them away can lead to a significant loss of precision. Hence, we do not explore this possibility in this paper.

4.3 Concretization by Composition of Shape Subgraphs

Intuitively, a shape subgraph represents the set of its super shape graphs. Concretization consists of connecting shape subgraphs such that the intersection of the sets of shape graphs that they represent is non-empty. To formalize this, we define the following binary relation on shape subgraphs.

Definition 4 (Subgraph Embedding). *We say that a shape subgraph* $G' \doteq (V', E', env', len')$ *is* embedded *in a shape subgraph* $G \doteq (V, E, env, len)$*, denoted* $G' \sqsubseteq G$*, if there exists a function* $f : V \to V'$ *such that: (i)* $(u, v) \in E$ *iff* $(f(u), f(v)) \in E'$*; (ii)* $f(env(x)) = env'(x)$ *for every* $x \in Vars(G)$*; and (iii) for every* $x \in Vars(G') \setminus Vars(G)$*,* $f^{-1}(env'(x)) \cap V = \emptyset$ *or* $env'(x) = env'(null)$*.*[6]

Thus, for any two atomic shape subgraphs G and G', $G' \sqsubseteq G$ iff $G = G'$.

We make $\langle SSG, \sqsubseteq\rangle$ a complete partial order by adding a special element $\perp$ to represent infeasible shape subgraphs, and define $\perp \sqsubseteq G$ for every shape subgraph G. We define the operation $compose : SSG \times SSG \to SSG$ that accepts two shape subgraphs and returns their greatest lower bound (w.r.t. to the $\sqsubseteq$ ordering). The operation naturally extends to sets of shape subgraphs.

Example 4. Referring to Fig. 2 and Fig. 3, we have $S_1 \sqsubseteq M_1$ and $S_1 \sqsubseteq M_4$, and $compose(M_1, M_4) = S_1$. □

The concretization function $\gamma^{GD} : 2^{ASSG} \to 2^{SG}$ is defined by

$$\gamma^{GD}(XG) \doteq \{G \mid G = compose(Y), Y \subseteq XG, \text{ G is admissible}\} .$$

This gives us the Galois Connection $\langle 2^{SG}, \alpha^{GD}, \gamma^{GD}, 2^{ASSG}\rangle$.

Properties of the Abstraction. Note that there is neither a loss of precision nor a gain in efficiency (e.g., such as a reduction in the size of the representation) when we decompose a single shape graph, i.e., $\gamma^{GD}(\beta^{GD}(G)) = \{G\}$. Both potentially appear when we abstract a *set of shape graphs* by decomposing each graph in a set. However, when there is no logical correlation between the different subgraphs (in the graph decomposition), we will gain efficiency without compromising precision.

Example 5. Consider the graphs in Fig. 2 and Fig. 3. Abstracting S_1 gives $\beta^{GD}(S_1) = \{M_1, M_4\}$. Concretizing back, gives $\gamma^{GD}(\{M_1, M_4\}) = \{S_1\}$. Abstracting S_5 yields $\beta^{GD}(S_5) = \{M_2, M_5\}$. Concretizing $\{M_1, M_2, M_4, M_5\}$ results in $\{S_1, S_2, S_4, S_5\}$, which overapproximates $\{S_1, S_5\}$. □

[6] We define $f^{-1}(x) \doteq \{y \in V \,.\, f(y) = x\}$.

5 Developing Efficient Abstract Transformers for the Graph Decomposition Abstraction

In this section, we show that it is hard to compute the most precise transformer for the graph decomposition abstraction in polynomial time and develop sound and efficient transformers. We demonstrate our ideas using the statement `t1.n=temp` in the running example and the subgraphs in Fig. 6 and Fig. 3.

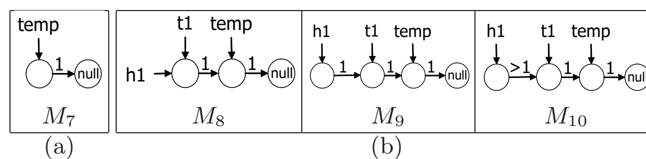


Fig. 6. (a) A subgraph at label `L2` in Fig. 1, and (b) Subgraphs at `L3` in Fig. 1

An abstract transformer $T_{st} : 2^{ASSG} \to 2^{ASSG}$ is *sound* for a statement *st* if for every set of shape subgraphs *XG* the following holds:

$$(\alpha^{GD} \circ [\![st]\!]^{\#} \circ \gamma^{GD})(XG) \subseteq T_{st}(XG) \ . \tag{1}$$

5.1 The Most Precise Abstract Transformer

We first show how the *most precise transformer* $[\![st]\!]^{GD} \doteq \alpha^{GD} \circ [\![st]\!]^{\#} \circ \gamma^{GD}$ can be computed *locally*, without concretizing complete shape graphs. As observed by Distefano et al. [3], the full heap abstraction transformer $[\![st]\!]^{\#}$ can be computed by considering only the *relevant* part of an abstract heap. We use this observation to create a local transformer for our graph decomposition abstraction.

The first step is to identify the subgraphs "referred" to by the statement *st*. Let $Vars(st)$ denote the variables that occur in statement *st*. We define:

- The function $modcomps_{st} : 2^{SSG} \to 2^{SSG}$ returns the shape subgraphs that have a variable in $Vars(st)$: $modcomps_{st}(XG) \doteq \{G \in XG \mid Vars(G) \cap Vars(st) \neq \emptyset\}$.
- The function $samecomps_{st} : 2^{SSG} \to 2^{SSG}$ returns the complementary subset: $samecomps_{st}(XG) \doteq XG \setminus modcomps_{st}(XG)$.

Example 6. $modcomps_{\texttt{t1.n=temp}}(\{M_1, \ldots, M_7\}) = \{M_1, M_2, M_3, M_7\}$ and $samecomps_{\texttt{t1.n=temp}}(\{M_1, \ldots, M_7\}) = \{M_4, M_5, M_6\}$. □

Note that the transformer $[\![st]\!]^{\#}$ operates on *complete* shape graphs. However, the transformer can be applied, in a straightforward fashion, to any shape *subgraph* G as long as G contains all variables mentioned in *st* (i.e., $Vars(G) \supseteq Vars(st)$). Thus, our next step is to compose subgraphs in $modcomps_{st}(XG)$ to generate subgraphs that contain all variables of *st*. However, not every set of subgraphs in $modcomps_{st}(XG)$ is a candidate for this composition step.

Given a set of subgraphs XG, a set $XG' \subseteq XG$, is defined to be *weakly feasible* in XG if $compose(XG') \neq \perp$. Further, we say that XG' is *feasible* in XG if there exists a subset $XR \subseteq XG$ such that $compose(XG' \cup XR)$ is an admissible shape graph (i.e., $\exists G \in SG : XG' \subseteq \alpha^{GD}(G) \subseteq XG$).

Example 7. The subgraphs M_1 and M_7 are feasible in $\{M_1, \ldots, M_7\}$, since they can be composed with M_4 to yield an admissible shape graph. However, M_1 and M_2 contain common variables and thus $\{M_1, M_2\}$ is not (even weakly) feasible in $\{M_1, \ldots, M_7\}$. In Fig. 7, the shape subgraphs M_1 and M_4 are weakly-feasible but not feasible in $\{M_1, \ldots, M_5\}$ (there is no way to compose subgraphs to include `w`, since M_1 and M_2 and M_3 and M_4 are not weakly-feasible.). □

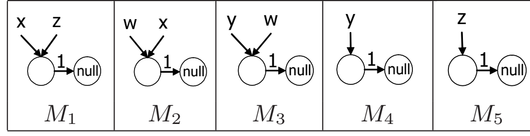


Fig. 7. A set of shape subgraphs over the set of program variables {`x`,`y`,`z`,`w`}

Let st be a statement with $k \doteq |Vars(st)|$ variables ($k \leq 2$ in our language). Let $M^{(\leq k)}$ denote all subsets of size k or less of a set M. We define the transformer for a heap-mutating statement st by:

$$\begin{aligned} T^{GD}_{st}(XG) \doteq \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid \; & M = modcomps_{st}(XG), R \in M^{(\leq k)}, \\ & G = compose(R), Vars(st) \subseteq Vars(G), \\ & R \text{ is feasible in } XG\} \\ \textbf{in } samecomps_{st}(XG) \cup \alpha^{GD}(Y) \; . & \end{aligned}$$

The transformer for an assume statement st is slightly different. An assume statement does not modify incoming subgraphs, but filters out some subgraphs that are not consistent with the condition specified in the assume statement. Note that it is possible for even subgraphs in $samecomps_{st}(XG)$ to be filtered out by the assume statement, as shown by the following definition of the transformer:

$$\begin{aligned} T^{GD}_{st}(XG) \doteq \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid \; & R \in XG^{(\leq k+1)}, \\ & G = compose(R), Vars(st) \subseteq Vars(G), \\ & R \text{ is feasible in } XG\} \\ \textbf{in } \alpha^{GD}(Y) \; . & \end{aligned}$$

Example 8. The transformer $T^{GD}_{\texttt{t1.n=temp}}$: (a) composes subgraphs: $compose(M_1, M_7)$, $compose(M_2, M_7)$, and $compose(M_3, M_7)$; (b) finds that the three pairs of subgraphs are feasible in $\{M_1, \ldots, M_7\}$; (c) applies the local full heap abstraction transformer $[\![\texttt{t1.n=temp}]\!]^{\#}$, producing M_8, M_9, and M_{10}, respectively; and (d) returns the final result: $T^{GD}_{\texttt{t1.n=temp}}(\{M_1, \ldots, M_7\}) = \{M_4, M_5, M_6\} \cup \{M_8, M_9, M_{10}\}$. □

Theorem 1. *The transformer T_{st}^{GD} is the most precise abstract transformer.*

Although T_{st}^{GD} applies $[\![st]\!]^{\#}$ to a polynomial number of shape subgraphs and $[\![st]\!]^{\#}$ itself can be computed in polynomial time, the above transformer is still exponential in the worst-case, because of the difficulty of checking the feasibility of R in XG. In fact, as we now show, it is impossible to compute the most precise transformer in *polynomial time*, unless P=NP.

Definition 5 (Most Precise Transformer Decision Problem). *The decision version of the most precise transformer problem is as follows: for a set of atomic shape subgraphs XG, a statement st, and an atomic shape subgraph G, does G belong to $[\![st]\!]^{GD}(XG)$?*

Theorem 2. *The most precise transformer decision problem, for the graph decomposition abstraction presented above, is NP-complete (even when the input set of subgraphs is restricted to be in the image of α^{GD}). Similarly, checking if XG' is feasible in XG is NP-complete.*

Proof (sketch). By reduction from the EXACT COVER problem: given a universe $U = \{u_1, \ldots, u_n\}$ of elements and a collection of subsets $A \subseteq 2^U$, decide whether there exists a subset $B \subseteq A$ such that every element $u \in U$ is contained in exactly one set in B. EXACT COVER is known to be NP-complete [4]. □

5.2 Sound and Efficient Transformers

We safely replace the check for whether R is feasible in XG by a check for whether R is weakly-feasible (i.e., whether $compose(R) \neq \perp$) and obtain the following transformer. (Note that a set of subgraphs is weakly-feasible iff no two of the subgraphs have a common variable; hence, the check for weak feasibility is easy.) For a heap-manipulating statement st, we define the transformer by:

$$\begin{aligned}\widehat{T_{st}^{GD}}(XG) \doteq \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid\ & M = modcomps_{st}(XG), R \in M^{(\leq k)}, \\ & G = compose(R) \neq \perp, Vars(st) \subseteq Vars(G)\} \\ \textbf{in } & samecomps_{st}(XG) \cup \alpha^{GD}(Y) \ .\end{aligned}$$

For an assume statement st, we define the transformer by:

$$\begin{aligned}\widehat{T_{st}^{GD}}(XG) \doteq \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid\ & R \in XG^{(\leq k+1)}, \\ & G = compose(R) \neq \perp, Vars(st) \subseteq Vars(G)\} \\ \textbf{in } & \alpha^{GD}(Y) \ .\end{aligned}$$

By definition, (1) holds for $\widehat{T_{st}^{GD}}$. Thus, $\widehat{T_{st}^{GD}}$ is a sound transformer.

We apply several engineering optimizations to make the transformer $\widehat{T_{st}^{GD}}$ efficient in practice: (i) by preceding statements of the form `x=y` and `x=y.n` with an assignment `x=null`, we specialize the transformer to achieve linear time complexity; (ii) we avoid unnecessary compositions of shape subgraphs for statements of the form `x.n=y` and `assume(x==y)`, when a shape subgraph contains both `x` and `y`; and (iii) assume statements do not change subgraphs, therefore we avoid performing explicit compositions and propagate atomic subgraphs.

5.3 An Incremental Transformer

The goal of an *incremental* transformer is to compute $\widehat{T^{GD}_{st}}(XG \cup \{D\})$ by reusing $\widehat{T^{GD}_{st}}(XG)$. We define the transformer for a heap-manipulating statement st by:

$$\begin{array}{ll}\widehat{T^{GD}_{st}}(XG \cup \{D\}) \doteq & \textbf{if } D \in \mathit{modcomps}_{st}(\{D\}) \\ & \quad \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid M = \mathit{modcomps}_{st}(XG \cup \{D\}), \\ & \qquad R \in M^{(\leq k)}, D \in R, \\ & \qquad G = \mathit{compose}(R) \neq \perp, \mathit{Vars}(st) \subseteq \mathit{Vars}(G)\} \\ & \quad \textbf{in } \widehat{T^{GD}_{st}}(XG) \cup \alpha^{GD}(Y) \\ & \textbf{else} \\ & \quad \widehat{T^{GD}_{st}}(XG) \cup \{D\} \ .\end{array}$$

Here, if the new subgraph D is not affected by the statement, we simply add it to the result. Otherwise, we apply the local full heap abstraction transformer only to subgraphs composed from the new subgraph (for sets of subgraphs not containing D, the result has been computed in the previous iteration).

For an assume statement st, we define the transformer by:

$$\begin{array}{ll}\widehat{T^{GD}_{st}}(XG \cup \{D\}) \doteq & \textbf{let } Y = \{[\![st]\!]^{\#}(G) \mid R \in (XG \cup \{D\})^{(\leq k+1)}, \\ & \qquad D \in R, G = \mathit{compose}(R) \neq \perp, \mathit{Vars}(st) \subseteq \mathit{Vars}(G)\} \\ & \textbf{in } \widehat{T^{GD}_{st}}(XG) \cup \alpha^{GD}(Y) \ .\end{array}$$

Again, we apply the transformer only to (composed) subgraphs containing D.

6 Prototype Implementation and Empirical Results

Implementation. We implemented the analyses based on the full heap abstraction and the graph decomposition abstraction described in previous sections in a system that supports memory deallocation and assertions of the form `assertAcyclicList(x)`, `assertCyclicList(x)`, `assertDisjointLists(x,y)`, and `assertReach(x,y)`. The analysis checks null dereferences, memory leakage, misuse of dangling pointers, and assertions. The system supports non-recursive procedure calls via call strings and unmaps variables as they become dead.

Example Programs. We use a set of examples to compare the full heap abstraction-based analysis with the graph decomposition-based analysis. The first set of examples consists of standard list manipulating algorithms operating on a single list (except for `merge`). The second set of examples consists of programs manipulating multiple lists: the running example, testing an implementation of a queue by two stacks[7], joining 5 lists, splitting a list into 5 lists, and two programs that model aspects of device drivers. We created the serial port driver example incrementally, first modeling 4 of the lists used by the device and then 5.

[7] `queue_2_stacks` was constructed to show a case where the graph decomposition-based analysis loses precision—determining that a queue is empty requires maintaining a correlation between the two (empty) lists.

Precision. The results of running the analyses appear in Tab. 2. The graph decomposition-based analysis failed to prove that the pointer returned by `getLast` is non-null[8], and that a dequeue operation is not applied to an empty queue in `queue_2_stacks`. On all other examples, the graph decomposition-based analysis has the same precision as the analysis based on the full heap abstraction.

Performance. The graph decomposition-based analysis is slightly less efficient than the analysis based on the full heap abstraction on the standard list examples. For the examples manipulating multiple lists, the graph decomposition-based analysis is faster by up to a factor of 212 (in the `serial_5_lists` example) and consumes considerably less space. These results are also consistent with the number of states generated by the two analyses.

7 Related Work

Single-graph Abstractions. Some early shape analyses used a single shape graph to represent the set of concrete states [8,1,16]. As noted earlier, it is possible to generalize our approach and consider different strategies for decomposing shape graphs. Interestingly, the single shape graph abstractions can be seen as one extreme point of such a generalized approach, which relies on a decomposition of a graph into its set of edges. The decomposition strategy we presented in this paper leads to a more precise analysis.

Partially Disjunctive Heap Abstraction. In previous work [12], we described a heap abstraction based on merging sets of graphs with the same set of nodes into one (approximate) graph. The abstraction in the current paper is based on decomposing a graph into a set of subgraphs. The abstraction in [12] suffers from the same exponential blow-ups as the full heap abstraction for our running example and examples containing multiple independent data structures.

Heap Analysis by Separation. Yahav et al. [18] and Hackett et al. [6] decompose heap abstractions to separately analyze different parts of the heap (e.g., to establish the invariants of different objects). A central aspect of the separation-based approach is that the analysis/verification problem is itself decomposed into a set of problem instances, and the heap abstraction is specialized for each problem instance and consists of one sub-heap consisting of the part of the heap relevant to the problem instance, and a coarser abstraction of the remaining part of the heap ([6] uses a points-to graph). In contrast, we simultaneously maintain abstractions of different parts of the heap and also consider the interaction between these parts. (E.g., it is possible for our decomposition to dynamically change as components get connected and disconnected.)

[8] A simple feasibility check while applying the transformer of the assertion would have eliminated the subgraph containing the null pointer.

Table 2. Time, space, number of states (shape graphs for the analysis based on full heap abstraction and subgraphs for the graph decomposition-based analysis), and number of errors reported. Rep. Err. and Act. Err. are the number of errors reported, and the number of errors that indicate real problems, respectively. #Loc indicates the number of CFG locations. F.H. and G.D. stand for full heap and graph decomposition, respectively

Benchmark		**Time** (sec.)		**Space** (Mb.)		**#States**		**R. Err./A. Err.**	
(**#Loc**)		**F.H.**	**G.D.**	**F.H.**	**G.D.**	**F.H.**	**G.D.**	**F.H.**	**G.D.**
`create`	(11)	0.03	0.19	0.3	0.3	27	36	0/0	0/0
`delete`	(25)	0.17	0.27	0.8	0.9	202	260	0/0	0/0
`deleteAll`	(12)	0.05	0.09	0.32	0.36	35	64	0/0	0/0
`getLast`	(13)	0.06	0.13	0.42	0.47	67	99	0/0	1/0
`getLast_cyclic`	(13)	0.08	0.09	0.39	0.41	53	59	0/0	0/0
`insert`	(23)	0.14	0.28	0.75	0.82	167	222	0/0	0/0
`merge`	(37)	0.34	0.58	2.2	1.7	517	542	0/0	0/0
`removeSeg`	(23)	0.19	0.33	0.96	1.0	253	283	0/0	0/0
`reverse`	(13)	0.09	0.12	0.47	0.46	82	117	0/0	0/0
`reverse_cyclic`	(14)	0.14	0.36	0.6	1.4	129	392	0/0	0/0
`reverse_pan`	(12)	0.2	0.6	0.9	2.2	198	561	0/0	0/0
`rotate`	(17)	0.05	0.08	0.3	0.4	33	50	0/0	0/0
`search_nulldref`	(7)	0.06	0.1	0.4	0.4	48	62	1/1	1/1
`swap`	(13)	0.05	0.09	0.3	0.4	35	62	0/0	0/0
`enqueueEvents`	(49)	0.2	0.2	1.2	0.7	248	178	0/0	0/0
`queue_2_stacks`	(61)	0.1	0.2	0.6	0.7	110	216	0/0	1/0
`join_5`	(68)	12.5	0.5	67.0	2.4	14,704	1,227	0/0	0/0
`split_5`	(47)	28.5	0.3	126.2	1.7	27,701	827	0/0	0/0
`1394diag`	(180)	26.2	1.8	64.7	8.5	10,737	4,493	0/0	0/0
`serial_4_lists`	(248)	36.9	1.7	230.1	11.7	27,851	6,020	0/0	0/0
`serial_5_lists`	(278)	552.6	2.6	849.2	16.4	89,430	7,733	0/0	0/0

Application to Other Shape Abstractions. Lev-Ami et al. [9] present an abstraction that could be seen as an extension of the full heap abstraction in this paper to more complex data structures, e.g., doubly-linked lists and trees. We believe that applying the techniques in this paper to their analysis is quite natural and can yield a more scalable analysis for more complex data structures. Distefano et al. [3] present a full heap abstraction based on separation logic, which is similar to the full heap abstraction presented in this paper. We therefore believe that it is possible to apply the techniques in this paper to their analysis as well. TVLA[10] is a generic shape analysis system that uses canonical abstraction. We believe it is possible to decompose logical structures in a similar way to decomposing shape subgraphs and extend the ideas in this paper to TVLA.

Decomposing Heap Abstractions for Interprocedural Analysis. Gotsman et al. [5] and Rinetzky et al. [14,15] decompose heap abstractions to create procedure summaries for full heap+ abstractions. This kind of decomposition, which does not lead to loss of precision (except when cutpoints are abstracted), is orthogonal to our decomposition of heaps, which is used to reduce the number of abstract states generated by the analysis. We believe it is possible to combine the two techniques to achieve a more efficient interprocedural shape analysis.

Acknowledgements. We thank Joseph Joy from MSR India for helpful discussions on Windows device drivers.

References

1. D. R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Proc. Conf. on Prog. Lang. Design and Impl.*, New York, NY, 1990. ACM Press.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, 1977. ACM Press, New York, NY.
3. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *In Proc. 13th Intern. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, 2006.
4. M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.
5. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Proceedings of the 13th International Static Analysis Symposium (SAS'06)*, 2006.
6. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proc. Symp. on Principles of Prog. Languages*, 2005.
7. N. D. Jones and S. S. Muchnick. Complexity of flow analysis, inductive assertion synthesis, and a language due to dijkstra. In *Program Flow Analysis: Theory and Applications*, chapter 12. Prentice-Hall, Englewood Cliffs, NJ, 1981.

8. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4. Prentice-Hall, Englewood Cliffs, NJ, 1981.
9. T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, 2006.
10. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symp.*, 2000.
11. R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. 2006. Full version.
12. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Proceedings of the 11th International Symposium, SAS 2004*, Lecture Notes in Computer Science. Springer, August 2004.
13. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*. Springer, January 2005.
14. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, 2005.
15. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th International Static Analysis Symposium (SAS)*, 2005.
16. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1), January 1998.
17. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.
18. E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004.

A Abstract Transformers for the Full Heap Abstraction

We now give the semantics of program statements over abstract states. Since the abstract domain does not introduce approximations by joining abstract elements, the transformers distribute over the shape graphs in a set. Thus, we can define each transformer for a single shape graph $G \doteq (V^G, E^G, env^G, len^G)$.

We define the following helper functions:

successor Returns the successor of a node.

$$\mathsf{successor}^G(v) = y \text{ s.t. } (x, y) \in E^G$$

focus If the edge outgoing from the given node is labeled by `>1` then the function splits the edge to represent the case where the uninterrupted list is of length 2 and the case where its length is greater than 2. Otherwise, it returns the original graph.

$$
\begin{array}{l}
\mathsf{focus}(G, v) \doteq \textbf{let } vn = \mathsf{successor}^G(v) \\
\quad \textbf{if} \quad len^G(v, vn) = 1 \\
\quad\quad XG_o \doteq \{G\} \\
\quad \textbf{else} \\
\quad\quad G_1 \doteq (V^{G_1}, E^{G_1}, env^{G_1}, len^{G_1}) \\
\quad\quad\quad V^{G_1} = V^G \cup \{m\} \\
\quad\quad\quad e_1 = (v, vn)\}, e_2 = (m, vn) \\
\quad\quad\quad E^{G_1} = E^G \setminus \{(v, vn)\} \cup \{e_1, e_2\} \\
\quad\quad\quad env^{G_1} = env^G \\
\quad\quad\quad len^{G_1} = len^G \setminus \{((v, vn), \cdot)\} \cup \{(e_1, 1), (e_2, \mathtt{1})\} \\
\quad\quad G_2 \doteq (V^{G_2}, E^{G_2}, env^{G_2}, len^{G_2}) \\
\quad\quad\quad V^{G_2} = V^G \cup \{m\} \\
\quad\quad\quad e_1 = (v, vn)\}, e_2 = (m, vn) \\
\quad\quad\quad E^{G_2} = E^G \setminus \{(v, vn)\} \cup \{e_1, e_2\} \\
\quad\quad\quad env^{G_2} = env^G \\
\quad\quad\quad len^{G_2} = len^G \setminus \{((v, vn), \cdot)\} \cup \{(e_1, 1), (e_2, \mathtt{>1})\} \\
\quad\quad XG_o \doteq \{G_1, G_2\}
\end{array}
$$

The transformers for each statement are shown below:

$$
\begin{array}{l}
T^{FH}_{\mathtt{x=new\ List()}}(G) \doteq \\
\quad \text{Update:} \\
\quad G' \doteq (V^{G'}, E^{G'}, env^{G'}, len^{G'}) \\
\quad\quad V^{G'} = V^G \cup \{v_{new}\} \text{ s.t. } v_{new} \notin V^G \\
\quad\quad e = (v_{new}, v^G_{null}) \\
\quad\quad E^{G'} = E^G \cup \{e\} \\
\quad\quad env^{G'} = \lambda z \in PVar \,.\, z = \mathtt{x} \; ? \; v_{new} \; : \; v^G_z \\
\quad\quad len^{G'} = len^G \cup \{(e, 1)\} \\
\quad G_o = \beta^{FH}(G')
\end{array}
$$

$T^{FH}_{\texttt{x=null}}(G) \doteq$

Update:

$G' \doteq (V^{G'}, E^{G'}, env^{G'}, len^{G'})$

$V^{G'} = V^G$

$E^{G'} = E^G$

$env^{G'} = \lambda z \in PVar \,.\, z = \texttt{x} \; ? \; v^G_{null} \; : \; v^G_z$

$len^{G'} = len^G$

$G_o = \beta^{FH}(G')$

$T^{FH}_{\texttt{x=y}}(G) \doteq$

Update:

$G' \doteq (V^{G'}, E^{G'}, env^{G'}, len^{G'})$

$V^{G'} = V^G$

$E^{G'} = E^G$

$env^{G'} = \lambda z \in PVar \,.\, z = \texttt{x} \; ? \; v^G_{\texttt{y}} \; : \; v^G_z$

$len^{G'} = len^G$

$G_o = \beta^{FH}(G')$

$T^{FH}_{\texttt{x=y.n}}(G) \doteq$

Precondition: $v^G_{\texttt{y}} \neq v^G_{null}$

Update:

$XG = \mathsf{focus}(G, v^G_{\texttt{y}})$

$XG_o = \emptyset$

foreach $G' = (V', E', env', len') \in XG$ {

$G_o \doteq (V^{G_o}, E^{G_o}, env^{G_o}, len^{G_o})$

$v_{\texttt{y.n}} = \mathsf{successor}^{G'}(v^{G'}_y)$

$V^{G_o} = V^{G'}$

$E^{G_o} = E^{G'}$

$len^{G_o} = len^{G'}$

$env^{G_o} = \lambda z \in PVar \,.\, z = \texttt{x} \; ? \; v_{\texttt{y.n}} \; : \; v^{G'}_z$

$XG_0 := XG \cup \{\beta^{FH}(G_o)\}$

}

$T^{FH}_{\texttt{x.n=y}}(G) \doteq$

Precondition: $v^G_{\texttt{x}} \neq v^G_{null}$

Update:

$G' \doteq (V^{G'}, E^{G'}, env^{G'}, len^{G'})$

$e_{kill} = (v^G_{\texttt{x}}, \mathsf{successor}^G(v^G_{\texttt{x}}))$

$e_{gen} = (v^G_{\texttt{x}}, v^G_{\texttt{y}})$

$V^{G'} = V^G$

$E^{G'} = E^G \setminus \{e_{kill}\} \cup \{e_{gen}\}$

$env^{G'} = env^G$

$len^{G'} = len^G \cup \{(e_{gen}, 1)\}$

$G_o = \beta^{FH}(G')$

B Algorithms for the Graph Decomposition Abstraction

B.1 Composing Shape Graphs

The following procedure gives a unique name for every node in a shape graph.

```
NameNodes(G) ≐
foreach variable var in G {
        int i := 0
        foreach node u ∈ V in DFS order
        starting from the node pointed-to by var {
                name(u) := name(u) ∪ {var_i}
                i := i + 1
        }
}

compose(G_1, G_2) ≐
        NameNodes(G_1)
        NameNodes(G_2)
        G = (V_1 ∪ V_2, E_1 ∪ E_2, nodelab_1 ∪ nodelab_2, len_1 ∪ len_2)
        if (consistent(G))
          return {G}
        else
          return ∅
```

B.2 Implementing Efficient Abstract Transformers

Let *st* be a program statement with variables x and y. The incremental abstract transformer for the statement accepts a shape graph G, a set of shape graphs XG (for which the transformer has already been computed, giving $\widehat{T_{st}^{GD}}(XG)$) and returns the delta: $\widehat{T_{st}^{GD}}(XG \cup \{G\}) \setminus \widehat{T_{st}^{GD}}(XG)$.

The algorithm associates with each input set XG a partitioning: $S_{st}(XG) \doteq XG_x, XG_y, XG_{same}$, such that: XG_x contains the graphs that have a node pointed-to by x but do not have a node pointed-to by y; XG_y contains the graphs that have a node pointed-to by y but do not have a node pointed-to by x; and XG_{same} contains the remaining graphs. The transformer defined below takes advantage of the partitioning and updates the auxiliary data structure S.

For a non-condition statement, the transformer is give by:

$$
\begin{aligned}
\widehat{T_{st}^{GD}}'(XG \cup \{G\}, S) \doteq\ & \textbf{if } x, y \notin G \\
& \qquad \textbf{return } (G, (XG_x, XG_y, XG_{same} \cup \{G\})) \\
& \textbf{if } x \in G \land y \notin G \\
& \qquad \textbf{let } Y = \{T_{st}^{FH}(G') \mid G' \in \mathit{compose}(M), \\
& \qquad\qquad M \in \{G\} \times G_y\} \\
& \textbf{in} \\
& \qquad \textbf{return } (G, (XG_x \cup \{G\}, XG_y, XG_{same})) \\
& \textbf{if } y \in G \land x \notin G \\
& \qquad \textbf{let } Y = \{T_{st}^{FH}(G') \mid G' \in \mathit{compose}(M), \\
& \qquad\qquad M \in \{G\} \times G_x\} \\
& \textbf{in} \\
& \qquad \textbf{return } (G, (XG_x, XG_y \cup \{G\}, XG_{same}))
\end{aligned}
$$

For a condition statement, the transformer is give by:

$$
\begin{aligned}
\widehat{T_{st}^{GD}}'(XG \cup \{G\}, S) \doteq\ & \textbf{if } x, y \notin G \\
& \qquad \textbf{let } Y = \{T_{st}^{FH}(G') \mid G' \in \mathit{compose}(M), \\
& \qquad\qquad M \in \{G\} \times G_y \times G_x\} \\
& \textbf{in} \\
& \qquad \textbf{return } (G, (XG_x \cup \{G\}, XG_y, XG_{same})) \\
& \textbf{if } x \in G \land y \notin G \\
& \qquad \textbf{let } Y = \{T_{st}^{FH}(G') \mid G' \in \mathit{compose}(M), \\
& \qquad\qquad M \in \{G\} \times G_y\} \\
& \textbf{in} \\
& \qquad \textbf{return } (G, (XG_x \cup \{G\}, XG_y, XG_{same})) \\
& \textbf{if } y \in G \land x \notin G \\
& \qquad \textbf{let } Y = \{T_{st}^{FH}(G') \mid G' \in \mathit{compose}(M), \\
& \qquad\qquad M \in \{G\} \times G_x\} \\
& \textbf{in} \\
& \qquad \textbf{return } (G, (XG_x, XG_y \cup \{G\}, XG_{same}))
\end{aligned}
$$

This transformer is more efficient than the one defined in Sec. 4 since it prunes shape graphs in XG that G could not possibly compose with.

C Experimental Results — Extra Details

Description of the example programs:

create Creates new elements and adds them to an acyclic list,
delete Deletes a cell chosen non-deterministically in an acyclic list,
deleteAll Deletes all elements of an acyclic list,
getLast Returns a pointer to the last element of an acyclic list,
getLast_cyclic Returns a pointer to the last element of a cyclic list,
insert Inserts an element to an acyclic list in a position chosen non-deterministically,
merge Merges two acyclic lists (simulates merging ordered lists),
removeSeg Removes a sublist from a cyclic list,
reverse Reverses an acyclic list,
reverse_cyclic Applying reversal to a cyclic list,
reverse_pan Applying reversal to a panhandle list,
rotate Moves the first element of an acyclic list to the tail,
search_nullderef A buggy implementation of a list search,
swap Swaps the first two elements of an acyclic list,
enqueueEvents The running example,
queue_2_stacks Test for an implementation of a queue using two lists,
join_5 A program joining 5 acyclic lists
split_5 A program that splits a list into 5 lists
1394diag Modeling aspects of the diagnostics program for the 1394 firewire device driver,
serial_4_lists Modeling aspects of 4 lists in the serial port device driver,
serial_5_lists Modeling aspects of 5 lists in the serial port device driver.

The code for the `queue_2_stacks` appears below.

```
// A procedure that tests an implementation of a queue via two stacks.
class List {
    public List n;
    public Object data;

    public List(Object data) {
        this.data = data;
    }
}

class Queue {
    List stack1;
    List stack2;

    public void qneueue(Object elem) {
        // Push into stack1.
        List cell = new List(elem);
        cell.n = stack1;
        stack1 = cell;
    }

    public Object dequeue() {
        if (isEmpty())
            throw IllegalOperationException();
        List cell;
        if (stack2 != null) {
            // Pop from stack2
            cell = stack2;
            stack2 = cell.n;
            // In C we would also free cell here.
            return cell.data;
        }
        else {
            // Pop contents of stack1 and push it to stack2.
            while (stack1 != null) {
                cell = stack1;
                stack1 = cell.n;
                cell.n = stack2;
                stack2 = cell;
            }
            // Now pop from stack2.
            cell = stack2;
            stack2 = cell.n;
            return cell.data; // In C we would also free cell here.
        }
    }

    public boolean isEmpty() { return stack1 == null && stack2 == null; }

    public static void main(String[] args) {
        Queue q = new Queue();
        if (...) {
            q.enqueue(1); // Now stack1 is not empty and stack2 is empty.
        }
        else {
            q.enqueue(2);
            q.enqueue(3);
            q.dequeue();
            // Now stack1 is empty and stack2 is not empty.
        }
        // At this point the non-relational abstraction
        // represents a state where both stacks are empty,
        // which causes a false alarm.
        q.dequeue();
    }
}
```